RSA BSAFE®

Crypto-C

Cryptographic Components for C

Library Reference Manual

Version 5.0

RSA SECURITY™

Contents

Chapter 1

Introduction

This manual is a reference guide for developers who use RSA BSAFE® Crypto-C (Crypto-C), a core-cryptography SDK.

To familiarize yourself with Crypto-C, you may wish to read the *Crypto-C User's Manual* before referencing this manual. In particular, you may be interested in the "Introductory Example" in Chapter 1. The *User's Manual* also describes cryptographic concepts and shows how to successfully add Crypto-C to your application.

After you have read the *User's Manual* and understand the steps involved in creating a Crypto-C application, you can turn to this reference manual. The algorithm information types (AIs) carry the underlying cryptography, so we put this essential information up front (Chapter 2). Next, the key information types (KIs) provide the values for those AIs that require keys (Chapter 3). The last chapter (Chapter 4) contains the application programming interface (API) itself. Make sure to read this chapter, as it provides a review of the Crypto-C components.

Organization

Chapter 1 introduces the components of the Crypto-C environment using a code example.

Chapters 2 and 3 are alphabetical listings of Crypto-C algorithm info types and key info types that specify the exact format of information supplied to and returned by Crypto-C.

Chapter 4 lists Crypto-C's functions alphabetically, giving full details of calling format and error return codes.

Appendix A lists Crypto-C error types.

Appendix B lists platform-specific types and constants.

Appendix C lists reference documents.

The Crypto-C Environment

The typical Crypto-C environment consists of five components:

- an application
- an algorithm chooser
- a surrender function
- the Crypto-C SDK
- memory management routines

A Crypto-C component is specific to either an application or a platform, or it is part of the Crypto-C SDK (see Figure 1-1).

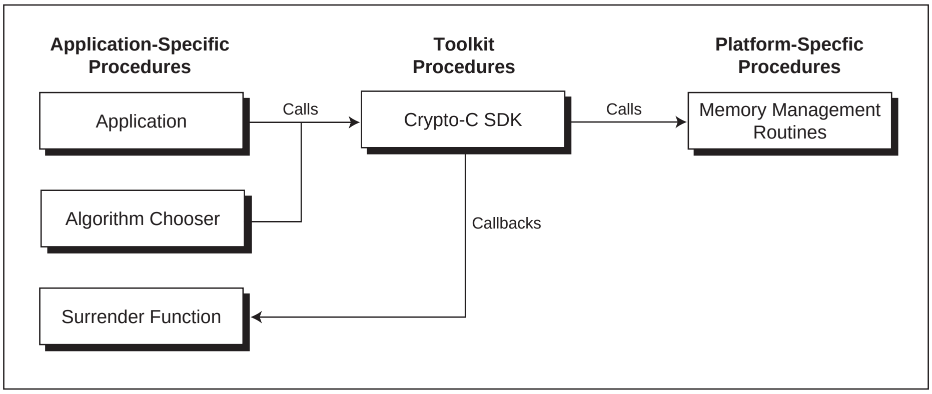


Figure 1-1 **The Crypto-C environment**

The application can be an encryption application, a key-generation application, or another similar application. The application calls Crypto-C and supplies the algorithm chooser and a callback to the surrender function. The algorithm chooser tells Crypto-C which method to use for a given algorithm. The surrender function allows processing or canceling during lengthy operations.

The Crypto-C SDK performs the cryptographic functions, such as encryption and key generation. The SDK is reentrant and suitable for shared or dynamic-link libraries.

Memory Management

Crypto-C provides memory management routines that perform memory allocation (`T_malloc`, `T_realloc` and `T_free`) and memory operations (`T_memcmp`, `T_memcpy`, `T_memmove`, and `T_memset`). These functions are modeled after conventional C library functions such as `malloc`, `memset`, etc. If you want to use the Crypto-C memory-management functions, you must link in the `tstdlib.c` file when you build your application.

You can also supply your own versions of these functions if, for example, you need platform-specific routines. In this case, you link to your application both your memory-management functions and the Crypto-C library. Crypto-C will use these statically linked functions in place of its memory management routines.

The BDEMO demonstration application provides a sample implementation of these functions. See "Details of Crypto-C Functions" on page 289 for descriptions and prototypes of these functions.

Code Example

The following code example summarizes the six-step model used in Crypto-C. This model is described in depth in Chapter 1 of the *User's Manual*. The example encrypts data within a DES key, using a function named `EncryptData`. The inputs to `EncryptData` are: a pointer to the buffer, the data's length, a pointer to the 8-byte DES key value, and a pointer to the 8-byte initialization vector used by the DES-CBC algorithm. `EncryptData` writes the encrypted data to the buffer supplied by the caller and returns the length of the encrypted data.

```
#include “aglobal.h”
#include “bsafe.h”
#include “demochos.h”
#define NULL_SURRENDER_PTR ((A_SURRENDER_CTX *)NULL_PTR)

int EncryptData
  (output, outputLen, maxOutputLen, input, inputLen, keyValue, iv)
unsigned char *output;                    /* pointer to output data */
unsigned int outputLen;       /* pointer to length of encrypted data */
unsigned int maxOutputLen;                     /* size of output buffer */
unsigned char *input;              /* pointer to input data buffer */
unsigned int inputLen;                        /* length of input data */
unsigned char *keyValue;                /* pointer to 8-byte DES key */
unsigned char *iv;       /* pointer to 8-byte initialization vector */

{
  B_ALGORITHM_OBJ desAlgorithm = (B_ALGORITHM_OBJ)NULL_PTR;
  B_KEY_OBJ desKey = (B_KEY_OBJ)NULL_PTR;
  B_BLK_CIPHER_W_FEEDBACK_PARAMS feedbackParams;
  ITEM initVector;
  unsigned int partOutLen;
  int status;

  /* break commands jump to the end of the do while (0) block */
  do {
    if ((status = B_CreateKeyObject (&desKey)) != 0)
      break;
    if ((status = B_SetKeyInfo
         (desKey, KI_DES8Strong, (POINTER)keyValue)) != 0)
      break;
    if ((status = B_CreateAlgorithmObject (&desAlgorithm)) != 0)
      break;
    initVector.data = iv;
    initVector.len = 8;
    feedbackParams.encryptionMethodName = “des”;
    feedbackParams.encryptionParams = NULL_PTR;
    feedbackParams.feedbackMethodName = “cbc”;
    feedbackParams.feedbackParams = initVector;
    feedbackParams.paddingMethodName = “pad”;
    feedbackParams.paddingParams = NULL_PTR;
    if ((status = B_SetAlgorithmInfo
         (desAlgorithm, (POINTER)&AI_FeedbackCipher,
          (POINTER)&feedbackParams)) != 0)
      break;

    if ((status = B_EncryptInit
         (desAlgorithm, desKey, DEMO_ALGORITHM_CHOOSER,
          NULL_SURRENDER_PTR)) != 0)
      break;
    if ((status = B_EncryptUpdate
         (desAlgorithm, output, outputLen, maxOutputLen, input,
          inputLen, (B_ALGORITHM_OBJ)NULL_PTR,
          NULL_SURRENDER_PTR)) != 0)
      break;

    if ((status = B_EncryptFinal
         (desAlgorithm, output + *outputLen, &partOutLen,
          maxOutputLen - *outputLen, (B_ALGORITHM_OBJ)NULL_PTR,
          NULL_SURRENDER_PTR)) != 0)
      break;
    *outputLen += partOutLen;
  } while (0);

  B_KEY_OBJ desKey = (B_KEY_OBJ)NULL_PTR;
  B_ALGORITHM_OBJ desAlgorithm = (B_ALGORITHM_OBJ)NULL_PTR;
  return (status);
}
```

The main work in *EncryptData* is done by the functions `B_EncryptInit`, `B_EncryptUpdate`, and `B_EncryptFinal`. The calling format for these and other functions is shown in Chapter 4, “Details of Crypto-C Functions.” Most Crypto-C procedures return either a zero for success or one of the error types listed in Appendix A.

`EncryptData` uses `DEMO_ALGORITHM_CHOOSER` as the *`algorithmChooser`* argument to `B_EncryptInit`. The algorithm chooser is described on page 10. `EncryptData` also uses `(A_SURRENDER_CTX *)NULL_PTR` as the *`surrenderContext`* argument to `B_EncryptInit`, `B_EncryptUpdate`, and `B_EncryptFinal`. As explained in Chapter 4, this tells Crypto-C to not call the surrender function. See the “The Surrender Function” on page 12 for an explanation of Crypto-C’s surrender routine.

The Algorithm Object

In the above code example, `B_EncryptInit`, `B_EncryptUpdate`, and `B_EncryptFinal` use an *algorithm object* called *`desAlgorithm`*. An algorithm object holds information about an algorithm's parameters (for example, the DES initialization vector), and keeps a context during a cryptographic operation (for example, DES encryption).

Before Crypto-C can use an algorithm object, you must create and set it with `B_CreateAlgorithmObject` and `B_SetAlgorithmInfo`.

Every algorithm object that is created by `B_CreateAlgorithmObject` must be destroyed by `B_DestroyAlgorithmObject`. For security reasons, when Crypto-C destroys an algorithm object, it zeroizes (sets to zero) and freezes any sensitive memory that the object allocated. Note that you can use an algorithm object for either encryption or decryption, but not for both. You must create separate algorithm objects to handle each case. Once you set an algorithm, do not reset it. In other words, once you call `B_SetAlgorithmInfo` for a particular algorithm object, do not call it again for the same object until that object has been destroyed and recreated.

As shown in Chapter 4, page 330, `B_SetAlgorithmInfo` has three input arguments, *`algorithmObject`*, *`infoType`*, and *`info.algorithmObject`* is the name of the algorithm object you are setting. *`infoType`* is one of the algorithm info types (AIs) listed in Chapter 2. The AI specifies which algorithm to use, such as DES-CBC, as well as the format of the actual algorithm information supplied by *`info`*.

As shown in Chapter 2, page 85, the format of *`info`* supplied to `B_SetAlgorithmInfo` for `AI_FeedbackCipher` is a pointer to a `B_BLK_CIPHER_W_FEEDBACK_PARAMS` structure that holds the necessary information for the encryption object. This data includes the encryption method name (“`des`”), the feedback method name (“`cbc`”), and a pointer to an `ITEM` structure that contains the 8 bytes of the initialization vector (*`iv`*), which seeds the process. In the preceding example, the data in the `ITEM` structure is the *`iv`* input argument to *`EncryptData`*.

The Key Object

In the above code example, `B_EncryptInit` uses a *key object* called *`desKey`*. A key object holds a key's value, such as the DES key, and supplies this value to a function, such as `B_EncryptInit`, that needs a key. A key object also receives the output of key generation such as `B_GenerateKeypair`.

Before Crypto-C can use a key object, you must create and set it with `B_CreateKeyObject` and `B_SetKeyInfo`. Every key object created by `B_CreateKeyObject` must be destroyed by `B_DestroyKeyObject`. For security reasons, when Crypto-C destroys a key object, it zeroizes (sets to zero) and freezes any sensitive memory that the object allocated. Once you call `B_SetKeyInfo` for a particular key object, do not call it again for the same object until it has been destroyed and recreated.

As shown in Chapter 4, page 331, `B_SetKeyInfo` has two input arguments, *`infoType`* and *`info`*. *`infoType`* is one of the KI key info types listed in Chapter 3. The key info type specifies the format of the actual key information supplied by *`info`*.

As shown in Chapter 3, the format of *`info`* supplied to `B_SetKeyInfo` for `KI_DES8Strong` is a pointer to an unsigned `char` array that holds the 8-byte DES key. In the code example on page 5, this is the *`keyValue`* input argument to *`EncryptData`*.

The Algorithm Chooser

The algorithm chooser lists all the algorithm methods that Crypto-C will use in the application. In this way, you only link in the code you need, and thereby reduce the executable size.

The BDEMO Algorithm Chooser

The BDEMO demonstration application supplied with Crypto-C defines an algorithm chooser called `DEMO_ALGORITHM_CHOOSER`. To use the BDEMO algorithm chooser, compile and link the module that defines `DEMO_ALGORITHM_CHOOSER` (as it is done for BDEMO) and specify `DEMO_ALGORITHM_CHOOSER` as the *`algorithmChooser`* argument.

Defining an Algorithm Chooser

The main reason for an application to define its own algorithm chooser is to make the executable image smaller. The linker links in the object code for all algorithm methods listed in the algorithm chooser. To illustrate, an application that only uses MD5 and DES-CBC may define an algorithm chooser with only the MD5 and DES-CBC methods. In this way, the linker will only link in the object code related to MD5 and DES-CBC.

An algorithm chooser is an array of pointers to `B_ALGORITHM_METHOD` values. The last element of the array must be `(B_ALGORITHM_METHOD *)NULL_PTR`. The following is an example that defines an algorithm chooser called `MD5_DES_CBC_CHOOSER` for the MD5 and DES-CBC algorithm methods needed when using the `AI_MD5WithDES_CBCPad` algorithm info type:

```
B_ALGORITHM_METHOD *MD5_DES_CBC_CHOOSER[] = {
  &AM_MD5,
  &AM_DES_CBC_ENCRYPT,
  &AM_DES_CBC_DECRYPT,
  (B_ALGORITHM_METHOD *)NULL_PTR
};
```

For additional examples of algorithm choosers, see “Algorithm Choosers” on page 118 of the *User’s Manual*.

Notice that encryption/decryption algorithm methods (AMs) such as DES-CBC, have separate entries for encryption and decryption. This separation of methods enables a minimal footprint. If an application only performs encryption, it may use an algorithm chooser with only the encrypt method to prevent the linker from linking in the object code for decryption.

Note: Early versions of Crypto-C offered optimized AMs for some algorithms on selected platforms. For instance, when employing RSA public encryption, there were: `AM_RSA_ENCRYPT`, `AM_RSA_ENCRYPT_68` for the MPW compiler on 68K machines, and `AM_RSA_ENCRYPT_86` for the Microsoft compiler on x86 platforms. In versions 3.0 and later, the optimized code for a particular algorithm, when available, is automatically linked in with the standard AM. So although it is no longer necessary to specify an optimized AM to link in optimized code, the old AM-specific optimized code is still valid.

The Surrender Function

During lengthy operations, such as public-key computations and key generation, Crypto-C surrenders control to the application's surrender function. The surrender function may notify users of the operation being performed, indicate that it is still performing, process operating system tasks, or execute other operations before returning to Crypto-C. The surrender function may also cause Crypto-C to cancel the operation by returning a non-zero value.

Your application specifies its surrender function through an A_SURRENDER_CTX value. A_SURRENDER_CTX is supplied as the *surrenderContext* argument to Crypto-C functions such as B_EncryptInit. A_SURRENDER_CTX is defined as follows:

```
typedef struct {
  int (*Surrender) ();                  /* surrender function callback */
  POINTER handle;                   /* application-specific information */
  POINTER reserved;                          /* reserved for future use */
} A_SURRENDER_CTX;
```

An A_SURRENDER_CTX value consists of a pointer to your application-specific callback function (that constitutes Crypto-C's surrender function) and a pointer to application-specific information. The pointer to application-specific information is supplied directly to your callback and is not otherwise manipulated by Crypto-C. The reserved value should be set to NULL_PTR for future compatibility.

A typical application initializes the A_SURRENDER_CTX value before calling a Crypto-C procedure. Each A_SURRENDER_CTX value may specify a different surrender function callback and a different handle.

For a sample surrender function, see "A Sample Surrender Function" in the *User's Manual*.

Surrender

The surrender function callback must have the following form:

```
int (*Surrender) (
  POINTER handle                     /* application-specific information */
);
```

Surrender is a developer-supplied function that actually performs the tasks required by the application. *handle* is the application-specific handle from the surrender context.

Surrender should return 0 for Crypto-C to continue its operation or a non-zero value for Crypto-C to cancel its operation.

Return value

0	continue with operation
non-zero	cancel operation

The ITEM Structure

Often, Crypto-C requires that input be in the form of an `ITEM` structure, defined as follows:

```
typedef struct {
  unsigned char *data;
  unsigned int   len;
} ITEM;
```

Chapter 2

Algorithm Info Types

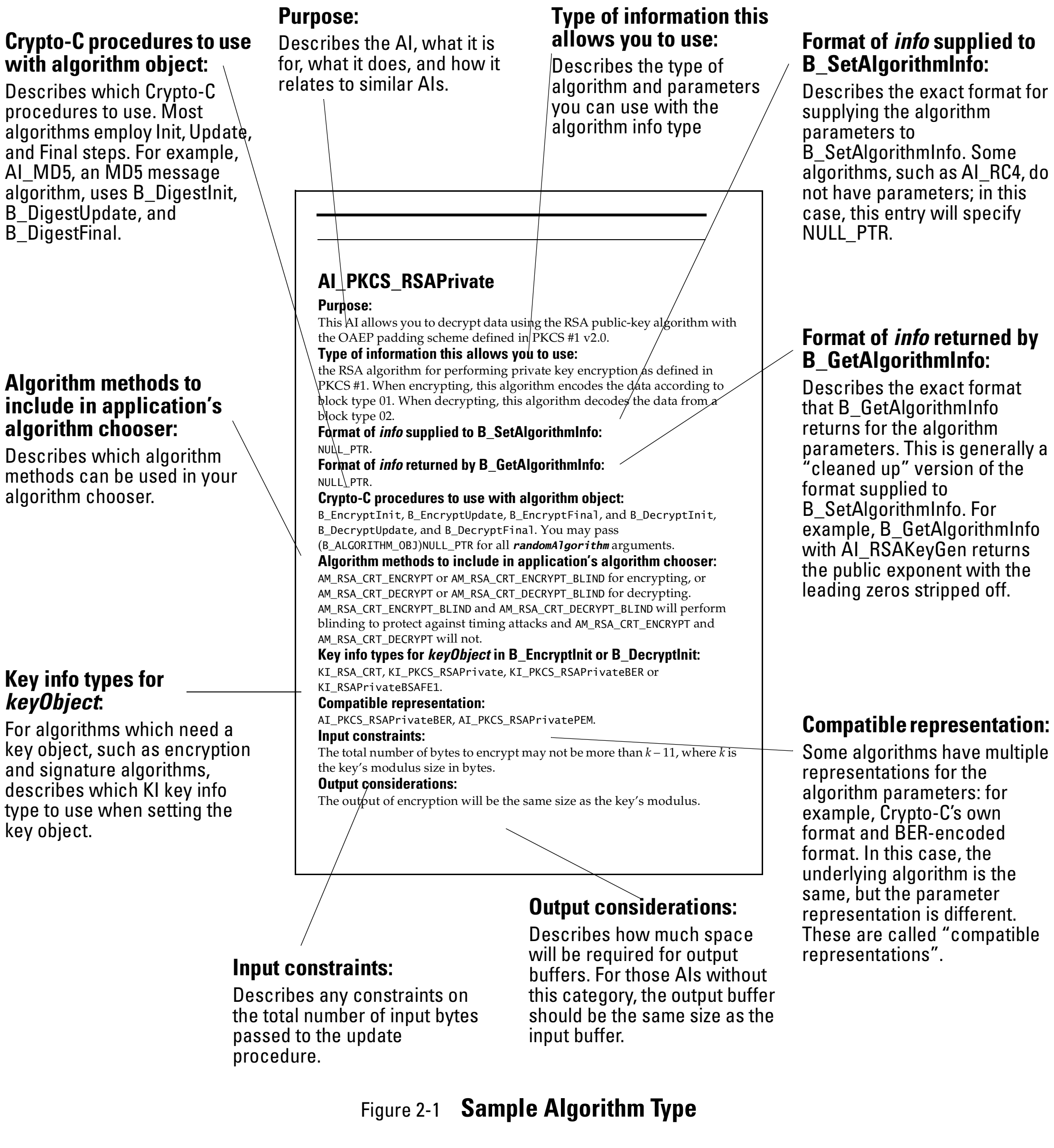
This chapter lists the standard algorithm info types (AIs) offered in RSA BSAFE Crypto-C (Crypto-C). A typical application supplies an algorithm info type as the *`infoType`* argument to `B_SetAlgorithmInfo`. For examples of how to use algorithm info types with certain algorithms, see the *User's Manual*.

An AI not only specifies which algorithm to use, but also specifies the format of the algorithm parameters. You supply the algorithm parameters as the *`info`* argument to `B_SetAlgorithmInfo`.

The entry for each AI occupies one or two pages. See the next page for a description of the entry format.

Some algorithm info types, such as encryption and signature algorithms, need a key object. If an algorithm calls for a key object, the AI entry will describe which KI to use to set the key object. For a complete list of KIs, see Chapter 3.

The algorithm info types pass information from and to various `B_` functions. See Chapter 4 for the descriptions and prototypes of these functions.



Crypto-C procedures to use with algorithm object:

Describes which Crypto-C procedures to use. Most algorithms employ Init, Update, and Final steps. For example, AI_MD5, an MD5 message algorithm, uses B_DigestInit, B_DigestUpdate, and B_DigestFinal.

Purpose:

Describes the AI, what it is for, what it does, and how it relates to similar AIs.

Type of information this allows you to use:

Describes the type of algorithm and parameters you can use with the algorithm info type

Format of *info* supplied to B_SetAlgorithmInfo:

Describes the exact format for supplying the algorithm parameters to B_SetAlgorithmInfo. Some algorithms, such as AI_RC4, do not have parameters; in this case, this entry will specify NULL_PTR.

AI_PKCS_RSAPrivate

Purpose:

This AI allows you to decrypt data using the RSA public-key algorithm with the OAEP padding scheme defined in PKCS #1 v2.0.

Type of information this allows you to use:

the RSA algorithm for performing private key encryption as defined in PKCS #1. When encrypting, this algorithm encodes the data according to block type 01. When decrypting, this algorithm decodes the data from a block type 02.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, and `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all ***randomAlgorithm*** arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_RSA_CRT_ENCRYPT` or `AM_RSA_CRT_ENCRYPT_BLIND` for encrypting, or `AM_RSA_CRT_DECRYPT` or `AM_RSA_CRT_DECRYPT_BLIND` for decrypting. `AM_RSA_CRT_ENCRYPT_BLIND` and `AM_RSA_CRT_DECRYPT_BLIND` will perform blinding to protect against timing attacks and `AM_RSA_CRT_ENCRYPT` and `AM_RSA_CRT_DECRYPT` will not.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_RSA_CRT`, `KI_PKCS_RSAPrivate`, `KI_PKCS_RSAPrivateBER` or `KI_RSAPrivateBSAFE1`.

Compatible representation:

`AI_PKCS_RSAPrivateBER`, `AI_PKCS_RSAPrivatePEM`.

Input constraints:

The total number of bytes to encrypt may not be more than $k - 11$, where k is the key's modulus size in bytes.

Output considerations:

The output of encryption will be the same size as the key's modulus.

Format of *info* returned by B_GetAlgorithmInfo:

Describes the exact format that B_GetAlgorithmInfo returns for the algorithm parameters. This is generally a "cleaned up" version of the format supplied to B_SetAlgorithmInfo. For example, B_GetAlgorithmInfo with AI_RSAKeyGen returns the public exponent with the leading zeros stripped off.

Algorithm methods to include in application's algorithm chooser:

Describes which algorithm methods can be used in your algorithm chooser.

Key info types for *keyObject*:

For algorithms which need a key object, such as encryption and signature algorithms, describes which KI key info type to use when setting the key object.

Compatible representation:

Some algorithms have multiple representations for the algorithm parameters: for example, Crypto-C's own format and BER-encoded format. In this case, the underlying algorithm is the same, but the parameter representation is different. These are called "compatible representations".

Output considerations:

Describes how much space will be required for output buffers. For those AIs without this category, the output buffer should be the same size as the input buffer.

Input constraints:

Describes any constraints on the total number of input bytes passed to the update procedure.

Figure 2-1 **Sample Algorithm Type**

AI_BSSecretSharing

Purpose:

This AI allows you to split a highly sensitive secret, such as a private key, into several "shares", which can be reassembled to recreate the original secret. The secret can only be recreated if there are at least a "threshold" number of shares present. For example, the secret can be divided into five shares. If the threshold is three, any three of them can be used to reconstruct the secret.

Type of information this allows you to use:

the Bloom-Shamir secret sharing algorithm as defined in "Generalized Linear Threshold Scheme" by S.C. Kothari, *Proceedings of CRYPTO 84*.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_SECRET_SHARING_PARAMS` structure:

```
typedef struct {
  unsigned int threshold;                        /* share threshold */
} B_SECRET_SHARING_PARAMS;
```

The *threshold* is the minimum number of shares required to recover the secret key; it has a minimum value of 2 and maximum of 255.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `B_SECRET_SHARING_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`.

`B_EncryptUpdate` must be called a minimum of *threshold* times. Each time, the secret being split must be supplied as the input and one new share is returned as the output. `B_EncryptFinal` returns a status of `BE_OUTPUT_COUNT` if the number of calls to `B_EncryptUpdate` calls is less than the *threshold*. `B_EncryptFinal` supplies no output.

You must supply an initialized random algorithm to `B_EncryptUpdate`. (The random algorithm is used only on the first call to `B_EncryptUpdate`). Supply `(B_ALGORITHM_OBJ)NULL_PTR` as the *randomAlgorithm* for `B_EncryptFinal`.

`B_DecryptUpdate` must be called *threshold* times to supply enough shares to recover the secret key. `B_DecryptFinal` returns a status of `BE_INPUT_COUNT` if the number of calls to `B_DecryptUpdate` is less than the *threshold*; otherwise, it returns a success status and outputs the secret key.

Supply (`B_ALGORITHM_OBJ`)`NULL_PTR` as the *randomAlgorithm* for `B_DecryptUpdate` and `B_DecryptFinal`. Supply (`B_KEY_OBJ`)`NULL_PTR` as the *keyObject* for `B_EncryptInit` and `B_DecryptInit`.

Output considerations:

The size of the output from each call to `B_EncryptUpdate` will be one byte more than the size of the secret. That byte represents the share.

AI_CBC_IV8

Purpose:

This AI allows you to change the initialization vector (IV) for a CBC-mode cipher without creating a new algorithm object or binding in a new key. This increases the performance of applications that have a long-lived symmetric key (for example, a DES key) used to encrypt many blocks or messages, each with a unique IV.

Type of information this allows you to use:

a new 8-byte initialization vector for an existing CBC algorithm object previously initialized with one of the following AIs:

`AI_DES_CBC_IV8`, `AI_DES_CBCPadIV8`, `AI_DES_CBCPadBER`, `AI_DES_CBCPadPEM`, `AI_DES_EDE3_CBC_IV8`, `AI_DES_EDE3_CBCPadIV8`, `AI_DES_EDE3_CBCPadBER`, `AI_DESX_CBC_IV8`, `AI_DESX_CBCPadIV8`, `AI_DESX_CBCPadBER`, `AI_RC2_CBC`, `AI_RC2_CBCPad`, `AI_RC2_CBCPadBER`, `AI_RC2_CBCPadPEM`, `AI_RC5_CBC`, and `AI_RC5_CBCPad`.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an 8-byte `unsigned char` array containing the new initialization vector.

This new initialization vector will not affect the current algorithm object until the next call to `B_EncryptInit`, `B_EncryptFinal`, `B_DecryptInit`, or `B_DecryptFinal`.

After the new IV takes effect on the algorithm object, any results from a previous call to `B_GetAlgorithmInfo` on the algorithm object are undefined.

Format of *info* returned by B_GetAlgorithmInfo:

`B_GetAlgorithmInfo` is not supported for `AI_CBC_IV8`.

Crypto-C procedures to use with algorithm object:

`B_SetAlgorithmInfo`.

AI_CBC_IV16

Purpose:

This AI allows you to change the initialization vector (IV) for a CBC-mode cipher without creating a new algorithm object or binding in a new key. This increases the performance of applications that have a long-lived symmetric key (for example, an RC6 key) used to encrypt many blocks or messages, each with a unique IV.

Type of information this allows you to use:

a new 16-byte initialization vector for an existing CBC algorithm object previously initialized with one of the following AIs:

`AI_RC6_CBC`, `AI_RC6_CBCPad`

Format of info supplied to B_SetAlgorithmInfo:

pointer to a 16-byte `unsigned char` array containing the new initialization vector. This new initialization vector will not affect the current algorithm object until the next call to `B_EncryptInit`, `B_EncryptFinal`, `B_DecryptInit`, or `B_DecryptFinal`. After the new IV takes effect on the algorithm object, any results from a previous call to `B_GetAlgorithmInfo` on the algorithm object are undefined.

Format of info returned by B_GetAlgorithmInfo:

`B_GetAlgorithmInfo` is not supported for `AI_CBC_IV16`.

Crypto-C procedures to use with algorithm object:

`B_SetAlgorithmInfo`.

AI_DES_CBC_BSAFE1

Purpose:

Deprecated. This AI is included only for backward compatibility.

Type of information this allows you to use:

the encryption type parameter (pad, pad with checksum, or raw) for the DES encryption algorithm as defined by BSAFE 1.x.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a B_BSAFE1_ENCRYPTION_PARAMS structure:

```
typedef struct {
  int encryptionType;                              /* encryption type */
} B_BSAFE1_ENCRYPTION_PARAMS;
```

encryptionType should be set to B_BSAFE1_PAD for pad mode, B_BSAFE1_PAD_CHECKSUM for pad with checksum mode, or B_BSAFE1_RAW for raw mode.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a B_BSAFE1_ENCRYPTION_PARAMS structure (see above).

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, B_EncryptFinal, B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. You may pass (B_ALGORITHM_OBJ)NULL_PTR for all *randomAlgorithm* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_DES_CBC_ENCRYPT for encryption and AM_DES_CBC_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_DES8Strong, KI_DES8, KI_8Byte, or KI_Item (if the length of the ITEM is 8).

Input constraints:

During encryption, this algorithm does not pad the output. Thus, you must provide input that is a multiple of 8 bytes.

Output considerations:

In pad mode, the total number of output bytes from encryption can be as many as 8 bytes more than the total input. In pad with checksum mode, the total number of output encryption bytes can be as many as 16 bytes more than the total input.

AI_DES_CBC_IV8

Purpose:

This AI allows you to perform DES encryption or decryption in CBC mode with an 8-byte initialization vector on data that is a multiple of 8 bytes long. No padding will be performed. See `AI_DES_CBCPadIV8` for the same algorithm type with padding.

Type of information this allows you to use:

an 8-byte initialization vector for the DES-CBC encryption algorithm as defined in FIPS PUB 46-1 and FIPS PUB 81.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `unsigned char` array that holds the 8 bytes of the initialization vector.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `unsigned char` array that holds the 8 bytes of the initialization vector.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_DES_CBC_ENCRYPT` for encryption and `AM_DES_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_DES8Strong`, `KI_DES8`, `KI_8Byte`, `KI_Item` (if the length of the `ITEM` is 8), or `KI_DES_BSAFE1`.

Input constraints:

During encryption, this algorithm does not pad the output. Thus, you must provide input that is a multiple of 8 bytes.

Token-based algorithm methods:

AI_DES_CBC_IV8 can be used to access the hardware-related algorithm methods AM_TOKEN_DES_CBC_ENCRYPT and AM_TOKEN_DES_CBC_DECRYPT, for use in conjunction with BHAPI.

Token-based key info types:

When used with one of the hardware algorithm methods described, AI_DES_CBC_IV8 should be used with KI_Token or KI_ExtendedToken.

AI_DES_CBCPadBER

Purpose:

This AI is similar to AI_DES_CBCPadIV8 except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call B_SetAlgorithmInfo to initialize an algorithm object from the encoded algorithm identifier, which includes the initialization vector. Alternatively, you call B_GetAlgorithmInfo with this AI to create an encoded algorithm identifier from an algorithm object created with AI_DES_CBCPadBER, AI_DES_CBCPadIV8, or AI_DES_CBCPadPEM. The OID for this algorithm—excluding the tag and length bytes—in decimal, is "43, 14, 3, 2, 7". Also see AI_DES_CBCPadIV8.

Type of information this allows you to use:

the encoded algorithm identifier that specifies the DES-CBC With Padding encryption algorithm as defined in FIPS PUB 46-1 and FIPS PUB 81, with padding scheme defined in PKCS #5 and desCBC algorithm identifier defined in [NIST91].

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an ITEM structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. B_SetAlgorithmInfo returns BE_WRONG_ALGORITHM_INFO if the algorithm identifier specifies an algorithm other than DES-CBC With Padding.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an ITEM structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, B_EncryptFinal, B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. You may pass (B_ALGORITHM_OBJ)NULL_PTR for all *randomAlgorithm* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_DES_CBC_ENCRYPT for encryption and AM_DES_CBC_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_DES8Strong, KI_DES8, KI_8Byte, KI_Item (if the length of the ITEM is 8), or KI_DES_BSAFE1.

Compatible representation:

AI_DES_CBCPadIV8, AI_DES_CBCPadPEM.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_DES_CBCPadIV8

Purpose:

This AI allows you to perform DES encryption or decryption in CBC mode with an 8-byte initialization vector on data that is of any byte length. The padding mode is PKCS #5, which makes the ciphertext 1 to 8 bytes longer than the plaintext. See `AI_DES_CBC_IV8` for the same algorithm type with no padding. See `AI_DES_CBCPadBER` for the same algorithm type with BER encoding.

Type of information this allows you to use:

an 8-byte initialization vector for the DES-CBC With Padding encryption algorithm as defined in FIPS PUB 46-1 and FIPS PUB 81, with padding scheme defined in PKCS #5.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `unsigned char` array that holds the 8 bytes of the initialization vector.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `unsigned char` array that holds the 8 bytes of the initialization vector.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_DES_CBC_ENCRYPT` for encryption and `AM_DES_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_DES8Strong`, `KI_DES8`, `KI_8Byte`, `KI_Item` (if the length of the `ITEM` is 8), or `KI_DES_BSAFE1`.

Compatible representation:

`AI_DES_CBCPadBER`, `AI_DES_CBCPadPEM`.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_DES_CBCPadPEM

Purpose:

This AI is similar to `AI_DES_CBCPadIV8` except that it uses the format defined in the Privacy Enhanced Mail protocol (PEM). This AI allows you to parse and create PEM algorithm identifiers. First, you call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier that includes the initialization vector. Alternatively, you call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object created using `AI_DES_CBCPadPEM`, `AI_DES_CBCPadIV8` or `AI_DES_CBCPadBER`. Also see `AI_DES_CBCPadIV8`.

Type of information this allows you to use:

an RFC 1423 identifier that specifies the DES-CBC With Padding encryption algorithm as defined in FIPS PUB 46-1 and FIPS PUB 81, with padding scheme defined in RFC 1423. This algorithm info type is intended to process the value of a DEK-Info field in a PEM encapsulated header.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a null-terminated string (`char *`) that gives the DES-CBC identifier and 8-byte initialization vector, for example, "`DES-CBC, 0123456789ABCDEF`". Space and tab characters are removed from the string before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an identifier other than DES-CBC.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a null-terminated string that gives the DES-CBC identifier and 8-byte initialization vector.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_DES_CBC_ENCRYPT` for encryption and `AM_DES_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_DES8Strong`, `KI_DES8`, `KI_8Byte`, `KI_Item` (if the length of the `ITEM` is 8) or `KI_DES_BSAFE1`.

Compatible representation:

`AI_DES_CBCPadIV8`, `AI_DES_CBCPadBER`.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_DES_EDE3_CBC_IV8

Purpose:

This AI allows you to perform three-key DES in encrypt-decrypt-encrypt mode as defined in ANSI X9.17 using the outer-CBC mode. This AI is initialized with an 8-byte IV and operates on data that is an exact multiple of 8 bytes long. No padding will be performed. See `AI_DES_EDE3_CBCPadIV8` for the same algorithm type with padding.

Type of information this allows you to use:

an 8-byte initialization vector for the DES-EDE3-CBC encryption algorithm.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `unsigned char` array that holds the 8 bytes of the initialization vector.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `unsigned char` array that holds the 8 bytes of the initialization vector.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_DES_EDE3_CBC_ENCRYPT` for encryption and `AM_DES_EDE3_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_DES24Strong`, `KI_24Byte`, or `KI_Item` (if the length of the `ITEM` is 24).

Input constraints:

During encryption, this algorithm does not pad the output. Thus, you must provide input that is a multiple of 8 bytes.

Token-based algorithm methods:

`AI_DES_EDE3_CBC_IV8` may be used to access the hardware-related algorithm methods `AM_TOKEN_DES_EDE3_CBC_ENCRYPT` and `AM_TOKEN_DES_EDE3_CBC_DECRYPT`, for use with BHAPI.

Token-based key info types:

When used with one of the hardware algorithm methods listed above, `AI_DES_EDE3_CBC_IV8` should be used with `KI_Token` or `KI_ExtendedToken`.

AI_DES_EDE3_CBCPadBER

Purpose:

This AI is similar to `AI_DES_EDE3_CBCPadIV8` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier that includes the initialization vector. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_DES_EDE3_CBC_PadIV8` or `AI_DES_EDE3_CBCPadBER`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 3, 7`". Also see `AI_DES_EDE3_CBCPadIV8`.

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the DES-EDE3-CBC encryption algorithm, with padding scheme defined in PKCS #5.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than DES-EDE3-CBC With Padding.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_DES_EDE3_CBC_ENCRYPT` for encryption and `AM_DES_EDE3_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_DES24Strong`, `KI_24Byte`, or `KI_Item` (if the length of the `ITEM` is 24).

Compatible representation:

`AI_DES_EDE3_CBCPadIV8`.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_DES_EDE3_CBCPadIV8

Purpose:

This AI allows you to perform three-key DES in encrypt-decrypt-encrypt mode as defined in ANSI X9.17 using the outer-CBC mode. This AI is initialized with an 8-byte IV and operates on data that is of any byte length. The padding mode is PKCS #5, which makes the ciphertext 1 to 8 bytes longer than the plaintext. See `AI_DES_EDE3_CBC_IV8` for the same algorithm type with no padding. See `AI_DES_EDE3_CBCPadBER` for the same algorithm type with BER encoding.

Type of information this allows you to use:

an 8-byte initialization vector for the DES-EDE3-CBC encryption algorithm, with padding scheme defined in PKCS #5.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `unsigned char` array that holds the 8 bytes of the initialization vector.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `unsigned char` array that holds the 8 bytes of the initialization vector.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_DES_EDE3_CBC_ENCRYPT` for encryption and `AM_DES_EDE3_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_DES24Strong`, `KI_24Byte`, or `KI_Item` (if the length of the `ITEM` is 24).

Compatible representation:

`AI_DES_EDE3_CBCPadBER`.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_DESX_CBC_BSAFE1

Purpose:

Deprecated. This AI is included only for backward compatibility.

Type of information this allows you to use:

the encryption type parameter (pad, pad with checksum, or raw) for the DESX encryption algorithm as defined by BSAFE 1.x.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a B_BSAFE1_ENCRYPTION_PARAMS structure:

```
typedef struct {
  int encryptionType;                              /* encryption type */
} B_BSAFE1_ENCRYPTION_PARAMS;
```

encryptionType should be set to B_BSAFE1_PAD for pad mode, B_BSAFE1_PAD_CHECKSUM for pad with checksum mode, or B_BSAFE1_RAW for raw mode.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a B_BSAFE1_ENCRYPTION_PARAMS structure (see above).

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, B_EncryptFinal, B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. You may pass (B_ALGORITHM_OBJ)NULL_PTR for all *randomAlgorithm* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_DESX_CBC_ENCRYPT for encryption and AM_DESX_CBC_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_DES24Strong, KI_24Byte, KI_Item (if the length of the ITEM is 24), KI_DESX or KI_DESX_BSAFE1.

Input constraints:

During encryption, this algorithm does not pad the output. Thus, you must provide input that is a multiple of 8 bytes.

Output considerations:

In pad mode, the total number of output bytes from encryption can be as many as 8 more than the total input. In pad with checksum mode, the total number of output bytes from encryption can be as many as 16 more than the total input.

AI_DESX_CBC_IV8

Purpose:

This AI allows you to perform DESX encryption or decryption in CBC mode with an 8-byte initialization vector on data that is a multiple of 8 bytes long. This algorithm takes 24 bytes of keying material. The first 8 bytes of the key form a standard 56-bit DES key, the second 8 bytes become the input whitening, and the last 8 bytes become the output whitening. The DESX algorithm has 64-bit input and output blocks like DES and it is used in all the same modes. Internally, the plaintext is XORed with the input whitening before running it through a DES encryption; and the output of DES is XORed with the output whitening to produce the output block of DESX. Decryption reverses those steps. See `AI_DESX_CBCPadIV8` for the same algorithm type with padding.

Type of information this allows you to use:

an 8-byte initialization vector for the DESX-CBC encryption algorithm, as defined by RSA Security Inc.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `unsigned char` array that holds the 8 bytes of the initialization vector.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `unsigned char` array that holds the 8 bytes of the initialization vector.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_DESX_CBC_ENCRYPT` for encryption and `AM_DESX_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_DES24Strong`, `KI_24Byte`, `KI_Item` (if the length of the `ITEM` is 24), `KI_DESX`, or `KI_DESX_BSAFE1`.

Input constraints:

During encryption, this algorithm does not pad the output. Thus, you must provide input that is a multiple of 8 bytes.

AI_DESX_CBCPadBER

Purpose:

This AI is similar to AI_DESX_CBCPadIV8 except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers, such as used in PKCS #7, and other protocols. You call B_SetAlgorithmInfo to initialize an algorithm object from the encoded algorithm identifier that includes the initialization vector. Alternatively, you call B_GetAlgorithmInfo with this AI to create an encoded algorithm identifier from an algorithm object that was created using AI_DESX_CBCPadIV8 or AI_DESX_CBCPadBER. The OID for this algorithm—excluding the tag and length bytes—in decimal, is "42, 134, 72, 134, 247, 13, 3, 6". Also see AI_DESX_CBCPadIV8.

Type of information this allows you to use:

the encoded algorithm identifier that specifies the DESX-CBC encryption algorithm, as defined by RSA Security Inc., with padding scheme defined in PKCS #5.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an ITEM structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. B_SetAlgorithmInfo returns BE_WRONG_ALGORITHM_INFO if the algorithm identifier specifies an algorithm other than DESX-CBC With Padding.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an ITEM structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, B_EncryptFinal, B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. You may pass (B_ALGORITHM_OBJ)NULL_PTR for all *randomAlgorithm* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_DESX_CBC_ENCRYPT for encryption and AM_DESX_CBC_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_DES24Strong`, `KI_24Byte`, `KI_Item` (if the length of the `ITEM` is 24), `KI_DESX`, or `KI_DESX_BSAFE1`.

Compatible representation:

`AI_DESX_CBCPadIV8`.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_DESX_CBCPadIV8

Purpose:

This AI allows you to perform DESX encryption or decryption in CBC mode. It is initialized with an 8-byte IV and operates on data that is any byte length. The padding mode is PKCS #5, which makes the ciphertext 1 to 8 bytes longer than the plaintext. See `AI_DESX_CBC_IV8` for the same algorithm type with no padding. See `AI_DESX_CBCPadBER` for the same algorithm type with BER encoding.

Type of information this allows you to use:

an 8-byte initialization vector for the DESX-CBC encryption algorithm, as defined by RSA Security Inc., with padding scheme defined in PKCS #5.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `unsigned char` array that holds the 8 bytes of the initialization vector.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `unsigned char` array that holds the 8 bytes of the initialization vector.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_DESX_CBC_ENCRYPT` for encryption and `AM_DESX_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_DES24Strong`, `KI_24Byte`, `KI_Item` (if the length of the `ITEM` is 24), `KI_DESX`, or `KI_DESX_BSAFE1`.

Compatible representation:

`AI_DESX_CBCPadBER`.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_DHKeyAgree

Purpose:

This AI allows you to perform Diffie-Hellman key agreement. You may have generated system parameters (for example, through AI_DHParamGen), or you may have retrieved them from another source. These system parameters are passed to B_SetAlgorithmInfo. The function B_KeyAgreePhase1 creates the public value that is sent to the other party, and B_KeyAgreePhase2 processes the value from the other party to produce the shared secret value. See AI_DHKeyAgreeBER for the same algorithm type with BER encoding.

Type of information this allows you to use:

Diffie-Hellman system parameters, where the prime and base integers, and the exponent size, are specified for performing Diffie-Hellman key agreement as defined in PKCS #3.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an A_DH_KEY_AGREE_PARAMS structure:

```
typedef struct {
  ITEM         prime;                               /* prime modulus */
  ITEM         base;                               /* base generator */
  unsigned int exponentBits;        /* size of random exponent in bits */
} A_DH_KEY_AGREE_PARAMS;
```

Each ITEM supplies an integer in canonical format, where the ITEM's *data* points to an unsigned byte array -- most significant byte first -- and the ITEM's *len* gives its length. All leading zeros are stripped from each integer before it is copied to the algorithm object.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an A_DH_KEY_AGREE_PARAMS structure (see above). All leading zeros have been stripped from each integer in the structure.

Crypto-C procedures to use with algorithm object:

`B_KeyAgreeInit`, `B_KeyAgreePhase1`, and `B_KeyAgreePhase2`. You must pass an initialized random algorithm to `B_KeyAgreePhase1`.

Algorithm methods to include in application's algorithm chooser:

`AM_DH_KEY_AGREE`.

Compatible representation:

`AI_DHKeyAgreeBER`.

Output considerations:

The size of the output of `B_KeyAgreePhase1` (the public value) will be the same size as the *`prime`*. The size of the output of `B_KeyAgreePhase2` (the agreed-upon secret) will also be the same size as the *`prime`*.

AI_DHKeyAgreeBER

Purpose:

This AI is similar to `AI_DHKeyAgree` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier that includes the prime modulus, base, and private value length. Alternatively, you call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_DHKeyAgree` or `AI_DHKeyAgreeBER`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 3, 1`". Also see `AI_DHKeyAgree`.

Type of information this allows you to use:

the encoded algorithm identifier that specifies Diffie-Hellman key agreement as defined in PKCS #3.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than Diffie-Hellman.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_KeyAgreeInit`, `B_KeyAgreePhase1`, and `B_KeyAgreePhase2`. You must pass an initialized random algorithm to `B_KeyAgreePhase1`.

Algorithm methods to include in application's algorithm chooser:

`AM_DH_KEY_AGREE`.

Compatible representation:

`AI_DHKeyAgree`.

Output considerations:

The size of the output of `B_KeyAgreePhase1` (the public value) will be the same size as the *`prime`*. The size of the output of `B_KeyAgreePhase2` (the agreed-upon secret) will also be the same size as the *`prime`*.

AI_DHParamGen

Purpose:

This AI allows you to generate Diffie-Hellman system parameters, which are the prime modulus, base, and private value length.

Type of information this allows you to use:

the parameters for generating Diffie-Hellman system parameters as defined in PKCS #3, where the size of the prime modulus and random exponent are specified. The optimized generating algorithm is proprietary, as defined by RSA Security Inc.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an A_DH_PARAM_GEN_PARAMS structure:

```
typedef struct {
  unsigned int primeBits;             /* size of prime modulus in bits */
  unsigned int exponentBits;        /* size of random exponent in bits */
} A_DH_PARAM_GEN_PARAMS;
```

The *exponentBits* must be less than *primeBits*.

Format of info returned by B_GetAlgorithmInfo:

pointer to an A_DH_PARAM_GEN_PARAMS structure (see above).

Crypto-C procedures to use with algorithm object:

B_GenerateInit and B_GenerateParameters. B_GenerateParameters sets the *resultAlgorithmObject* with the AI_DHKeyAgree information. You must pass an initialized random algorithm to B_GenerateParameters.

Algorithm methods to include in application's algorithm chooser:

AM_DH_PARAM_GEN.

AI_DSA

Purpose:

This AI allows you to create or verify raw DSA signatures when the 20-byte input is already known. It does not compute a message digest before applying the signature operation. See `AI_DSAWithSHA1` for the DSA algorithm type that involves the SHA1 digest operation.

Type of information this allows you to use:

the DSA signature algorithm for performing raw DSA signing and verifying as defined in FIPS PUB 186.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_SignInit`, `B_SignUpdate`, `B_SignFinal`, `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal`. You must pass a random algorithm in `B_SignFinal`, but may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all other *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_DSA_SIGN` for signature creation and `AM_DSA_VERIFY` for signature verification.

Key info types for *keyObject* in B_SignInit:

`KI_DSAPrivate`, `KI_DSAPrivateBER`, or `KI_DSAPrivateX957BER`.

Key info types for *keyObject* in B_VerifyInit:

`KI_DSAPublic`, `KI_DSAPublicBER`, or `KI_DSAPublicX957BER`.

Input constraints:

The DSA algorithm requires that the input data (the data to sign) be exactly 20 bytes long. Normally, this 20-byte input is the result of SHA1 output.

Output considerations:

The *`signature`* result of `B_SignFinal` will be 40 bytes long; these bytes appear in the order (*r*, *s*).

Token-based algorithm methods:

`AI_DSA` may be used to access the hardware-related algorithm methods `AM_TOKEN_DSA_SIGN` and `AM_TOKEN_DSA_VERIFY` for use with BHAPI.

Token-based key info types:

When used with one of the hardware algorithm methods listed above, `AI_DSA` should be used with `KI_Token` or `KI_KeypairToken`.

AI_DSAKeyGen

Purpose:

This AI allows you to generate a DSA key pair. First, you pass the system parameters to `B_SetAlgorithmInfo`. Then you generate the keys by calling `B_GenerateInit` and `B_GenerateKeypair`. Alternatively, you may use the `AI_DSAParamGen` algorithm type to generate the system parameters needed in DSA key generation. Also see `AI_DSAParamGen`.

Type of information this allows you to use:

the parameters for generating a compatible DSA key pair as defined in FIPS PUB 186.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `A_DSA_PARAMS` structure:

```
typedef struct {
  ITEM prime;                                                /* the prime p */
  ITEM subPrime;                                          /* the subprime q */
  ITEM base;                                                  /* the base g */
} A_DSA_PARAMS;
```

An `ITEM` supplies an integer in canonical format, where the `ITEM`'s *`data`* points to an unsigned byte array -- most significant byte first -- and the `ITEM`'s *`len`* gives its length. All leading zeros are stripped from the integer before it is copied to the algorithm object.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `A_DSA_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_GenerateInit` and `B_GenerateKeypair`. `B_GenerateKeypair` sets the *`publicKey`* key object with the `KI_DSAPublic` information and the *`privateKey`* key object with the `KI_DSAPrivate` information. You must pass an initialized random algorithm to `B_GenerateKeypair`.

Algorithm methods to include in application's algorithm chooser:

`AM_DSA_KEY_GEN`.

AI_DSAParamGen

Purpose:

This AI allows you to generate DSA system parameters. The sizes of the parameters are passed to `B_SetAlgorithmInfo` and the parameters are made by calling `B_GenerateInit` and `B_GenerateParameters`. You use DSA parameters generated by this AI to generate a DSA key pair. Also see `AI_DSAKeyGen`.

Type of information this allows you to use:

the number of prime bits for generating a prime, a subprime, and a base (*p*, *q*, and *g*) compatible with FIPS PUB 186.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_DSA_PARAM_GEN_PARAMS` structure:

```
typedef struct {
  unsigned int primeBits;                    /* size of prime in bits */
} B_DSA_PARAM_GEN_PARAMS;
```

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `B_DSA_PARAM_GEN_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_GenerateInit` and `B_GenerateParameters`. `B_GenerateParameters` sets the *`resultAlgorithmObject`* algorithm object with the `AI_DSAKeyGen` information. You must pass an initialized random algorithm to `B_GenerateParameters`.

Algorithm methods to include in application's algorithm chooser:

`AM_DSA_PARAM_GEN`.

Notes:

The size of the subprime is always 160 bits.

AI_DSAWithSHA1

Purpose:

This AI allows you to create or verify SHA1 DSA signatures. It is passed the plaintext and computes the SHA1 digest as well as the DSA signature of that digest. See `AI_DSA` for the DSA algorithm type without the SHA1 digest operation. See `AI_DSAWithSHA1_BER` for the same algorithm type with BER encoding.

Type of information this allows you to use:

the DSA With SHA1 signature algorithm that uses the SHA1 digest algorithm and DSA to create and verify DSA digital signatures as defined in X9.57 Draft Section 5.3.1 and FIPS PUB 186.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_SignInit`, `B_SignUpdate`, `B_SignFinal`, `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal`. You must pass a random algorithm in `B_SignFinal`, but may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all other *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_SHA` and `AM_DSA_SIGN` for signature creation, and `AM_DSA_VERIFY` for signature verification.

Key info types for *keyObject* in B_SignInit:

`KI_DSAPrivate`, `KI_DSAPrivateBER`, or `KI_DSAPrivateX957BER`.

Key info types for *keyObject* in B_VerifyInit:

`KI_DSAPublic`, `KI_DSAPublicBER`, or `KI_DSAPublicX957BER`.

Compatible representation:

`AI_DSAWithSHA1_BER`.

Output considerations:

The *`signature`* result of `B_SignFinal` is a BER-encoded value of type SEQUENCE (INTEGER, INTEGER), where the first field is the value *r* and the second field is the value *s* as defined in section 5.3.1 of X9.57 Draft. The size of *`signature`* may be as many as 48 bytes.

AI_DSAWithSHA1_BER

Purpose:

This AI is similar to AI_DSAWithSHA1 except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as used in PKCS #7 and other protocols. You call B_SetAlgorithmInfo to initialize an algorithm object from the encoded algorithm identifier. You call B_GetAlgorithmInfo with this AI to create an encoded algorithm identifier from an algorithm object that was created using AI_DSAWithSHA1 or AI_DSAWithSHA1_BER. The OID for this algorithm, excluding the tag and length bytes, in decimal, is "43, 14, 3, 2, 27". Also see AI_DSAWithSHA1.

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the DSA With SHA1 signature algorithm that uses the SHA1 digest algorithm and DSA to create and verify DSA digital signatures as defined in X9.57 Draft Section 5.3.1 and FIPS PUB 186.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an ITEM structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. B_SetAlgorithmInfo returns BE_WRONG_ALGORITHM_INFO if the algorithm identifier specifies an algorithm other than DSA With SHA1.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an ITEM structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

B_SignInit, B_SignUpdate, B_SignFinal, B_VerifyInit, B_VerifyUpdate, and B_VerifyFinal. You must pass a random algorithm in B_SignFinal, but may pass (B_ALGORITHM_OBJ)NULL_PTR for all other *randomAlgorithm* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_SHA1 and AM_DSA_SIGN for signature creation, and AM_DSA_VERIFY for signature verification.

Key info types for *keyObject* in B_SignInit:

`KI_DSAPrivate`, `KI_DSAPrivateBER`, or `KI_DSAPrivateX957BER`.

Key info types for *keyObject* in B_VerifyInit:

`KI_DSAPublic`, `KI_DSAPublicBER`, or `KI_DSAPublicX957BER`.

Compatible representation:

`AI_DSAWithSHA1`.

Output considerations:

The *`signature`* result of `B_SignFinal` is a BER-encoded value of type SEQUENCE (INTEGER, INTEGER) where the first field is the value *r* and the second field is the value *s* as defined in section 5.3.1 of X9.57 Draft. The size of *`signature`* may be as many as 48 bytes.

AI_ECAcceleratorTable

Purpose:

This AI allows you to use the acceleration table for various elliptic curve (EC) operations that include encryption, signature creation and verification, key pair generation, and key agreement operations.

Type of information this allows you to use:

the acceleration table produced by the operation of `AI_ECBuildAcceleratorTable` on EC parameters or the table produced by the operation of `AI_ECBuildPubKeyAccelTable` on EC parameters.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` holding the accelerator table.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` holding the accelerator table.

Crypto-C procedures to use with algorithm object:

You use the table generated by `AI_ECBuildAcceleratorTable` with `B_SetAlgorithmInfo` to accelerate EC encryption in `AI_EC_ES`; signature creation and verification in `AI_EC_DSA` and `AI_EC_DSAWithDigest`; key pair generation in `AI_ECKeyGen`; and phase 1 of key agreement operations in `AI_EC_DHKeyAgree`. Use the table generated by `AI_ECBuildPubKeyAccelTable` with `B_SetAlgorithmInfo` to accelerate verification in `AI_EC_DSA` and `AI_EC_DSAWithDigest`.

Algorithm methods to include in application's algorithm chooser:

None.

AI_ECBuildAcceleratorTable

Purpose:

This AI allows you to build an acceleration table for the retrieval of a base point in various elliptic curve (EC) operations. To build a table that includes public key values as well as base point values, see `AI_ECBuildPubKeyAccelTable`.

Type of information this allows you to use:

elliptic curve parameters as defined in X9.62 to generate auxiliary data for the acceleration of EC operations with base point. Elliptic curve parameters can be generated through the execution of `AI_ECParamGen`.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_EC_PARAMS` structure:

```
typedef struct {
  B_INFO_TYPE parameterInfoType;        /* used to interpret EC parameters */
  POINTER     parameterInfoValue               /* describes elliptic curve */
} B_EC_PARAMS;
```

where *`parameterInfoType`* must be `AI_ECParameters` and *`parameterInfoValue`* must be an A_EC_PARAMS structure:

```
typedef struct {
  unsigned int version;                          /* implementation version */
  unsigned int fieldType;                  /* indicates type of base field */
  ITEM         fieldInfo;                         /*  It is the prime number */
                                                    /* if fieldType = FT_FP; */
                  /* the basis polynomial if fieldType = FT_F2_POLYNOMIAL; */
                   /* and the degree of the field if fieldType = FT_F2_ONB */
  ITEM         coeffA;                       /* elliptic curve coefficient */
  ITEM         coeffB;                       /* elliptic curve coefficient */
  ITEM         base;                    /* elliptic curve group generator */
  ITEM         order;           /* order of subgroup's generating element */
  ITEM         cofactor;                   /* the cofactor of the subgroup */
  unsigned int pointRepresentation;  /* not used. */
                                     /* set to CI_NO_COMPRESS as a default */
  unsigned int fieldElementBits;             /* field element size in bits */
} A_EC_PARAMS;
```

Format of *info* returned by B_GetAlgorithmInfo:

B_GetAlgorithmInfo is not supported with this AI. If called, it will return an error.

Crypto-C procedures to use with algorithm object:

B_BuildTableInit and B_BuildTableFinal.

Algorithm methods to include in application's algorithm chooser:

AM_ECFP_BLD_ACCEL_TABLE for odd prime fields and AM_ECF2POLY_BLD_ACCEL_TABLE for even characteristic.

Output Considerations:

The size of the accelerator table may be found through a call to B_BuildTableGetBufSize after a call to B_BuildTableInit.

AI_ECBuildPubKeyAccelTable

Purpose:

This AI allows you to build an acceleration table for a public key and the base point to be used in various elliptic curve (EC) operations. The acceleration values come from a table that includes values built for the base point, which are the same as those from `AI_ECBuildAcceleratorTable`, and additional values built for the public key. This AI performs at greater speeds than `AI_ECBuildAcceleratorTable` for both ECDSA verify and ECDH Phase 2 operations.

Type of information this allows you to use:

elliptic curve parameters as defined in X9.62 to generate auxiliary data for the acceleration of elliptic curve operations with base point and public key. Elliptic curve parameters can be generated through the execution of `AI_ECParamGen`.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_EC_PARAMS` structure:

```
typedef struct {
  B_INFO_TYPE parameterInfoType;        /* used to interpret EC parameters */
  POINTER     parameterInfoValue               /* describes elliptic curve */
} B_EC_PARAMS;
```

where *`parameterInfoType`* must be `AI_ECPubKey` and *`parameterInfoValue`* must be an `A_EC_PUBLIC_KEY` structure:

```
typedef struct {
  ITEM        publicKey;                                /* public component */
  A_EC_PARAMS curveParams;     /* the underlying elliptic curve parameters */
} A_EC_PUBLIC_KEY;
```

Format of *info* returned by B_GetAlgorithmInfo:

`B_GetAlgorithmInfo` is not supported with this AI. If called, it will return an error.

Crypto-C procedures to use with algorithm object:

`B_BuildTableInit` and `B_BuildTableFinal`.

Algorithm methods to include in application's algorithm chooser:

`AM_ECFP_BLD_PUB_KEY_ACCEL_TABLE` for odd prime fields and
`AM_ECF2POLY_BLD_PUB_KEY_ACCEL_TABLE` for even characteristic.

Output Considerations:

The size of the accelerator table may be found through a call to
`B_BuildTableGetBufSize` after a call to `B_BuildTableInit`.

AI_EC_DHKeyAgree

Purpose:

This AI allows you to perform elliptic curve Diffie-Hellman key agreement for given EC parameters. You may have generated system parameters (for example, through `AI_ECParamGen`), or you may have retrieved them from another source. These system parameters are passed to `B_SetAlgorithmInfo`. The function `B_KeyAgreePhase1` creates the public value that is sent to the other party, and `B_KeyAgreePhase2` processes the value from the other party to produce the shared secret value.

Type of information this allows you to use:

elliptic curve system parameters used in the elliptic curve, Diffie-Hellman key agreement operation, as defined in X9.63 Draft.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_EC_PARAMS` structure:

```
typedef struct {
  B_INFO_TYPE *parameterInfoType;       /* used to interpret EC parameters */
  POINTER      parameterInfoValue;             /* describes elliptic curve */
} B_EC_PARAMS;
```

where *parameterInfoType* must be `AI_ECParameters` and *parameterInfoValue* must be an A_EC_PARAMS structure:

```
typedef struct {
  unsigned int version;                          /* implementation version */
  unsigned int fieldType;                  /* indicates type of base field */
  ITEM         fieldInfo;       /* The prime number if fieldType = FT_FP; */
                  /* the basis polynomial if fieldType = FT_F2_POLYNOMIAL; */
                   /* and the degree of the field if fieldType = FT_F2_ONB */
  ITEM         coeffA;                      /* elliptic curve coefficient */
  ITEM         coeffB;                      /* elliptic curve coefficient */
  ITEM         base;                   /* elliptic curve group generator */
  ITEM         order;          /* order of subgroup's generating element */
  ITEM         cofactor;                 /* the cofactor of the subgroup */
  unsigned int pointRepresentation;          /* not used. */
                                     /* set to CI_NO_COMPRESS as a default */
  unsigned int fieldElementBits;             /* field element size in bits */
} A_EC_PARAMS;
```

Format of *info* returned by B_GetAlgorithmInfo:

B_GetAlgorithmInfo is not supported with this AI. If called, it will return an error.

Crypto-C procedures to use with algorithm object:

B_KeyAgreeInit, B_KeyAgreePhase1, and B_KeyAgreePhase2. You must pass an initialized random algorithm to B_KeyAgreePhase1.

Algorithm methods to include in application's algorithm chooser:

AM_ECFP_DH_KEY_AGREE for odd prime fields and AM_ECF2POLY_DH_KEY_AGREE for even characteristic.

Output considerations:

The size of Phase 1 output is 1 + 2 · (size of field element) bytes; the size of Phase 2 output is (size of field element) bytes

AI_EC_DSA

Purpose:

This AI allows you to perform raw ECDSA signature creation or verification operations. It does not compute a message digest before applying the signature operation. To compute a SHA1 message digest and create a signature of that digest, see `AI_EC_DSAWithDigest`.

Type of information this allows you to use:

the ECDSA signature algorithm used in raw ECDSA signature generation and verification, as defined in X9.62. Alternatively, to use an acceleration table in the generation or verification of a signature, use `AI_ECBuildAcceleratorTable` or `AI_ECBuildPubKeyAccelTable`. The public key-specific acceleration table accelerates verification only; for this operation, it provides greater acceleration than the `AI_ECBuildAcceleratorTable` at the cost of greater memory usage.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_SignInit`, `B_SignUpdate`, `B_SignFinal`, `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal`. You must pass a random algorithm in `B_SignFinal`, but may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all other *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

For signature creation, `AM_ECFP_DSA_SIGN` for odd prime fields and `AM_ECF2POLY_DSA_SIGN` for even characteristic. For signature verification, `AM_ECFP_DSA_VERIFY` for odd prime fields and `AM_ECF2POLY_DSA_VERIFY` for even characteristic.

Key info types for keyObject in B_SignInit:

`KI_ECPrivate` or `KI_ECPrivateBER`.

Key info types for keyObject in B_VerifyInit:

`KI_ECPublic` or `KI_ECPublicBER`.

Input constraints:

In practice, the input to the ECDSA algorithm — that is, the data to sign — is generally the result of a digest operation. In Crypto-C's implementation, however, the only restrictions on the input are that it must be at least 16 bytes and no more than 32 bytes long.

Output considerations:

The *`signature`* result of `B_SignFinal` is the concatenation of the signature elements (r,s), as defined in section 5.3.1 of the X9.57 Draft. The size of *`signature`* is 2*(length of order) bytes. For instance, if the order is 160 bits (20 bytes), the *`signature`* will be 40 bytes long.

AI_EC_DSAWithDigest

Purpose:

This AI allows you to create or verify SHA1 ECDSA signatures. It is passed the plaintext and computes the SHA1 digest as well as the ECDSA signature of that digest. See `AI_EC_DSA` for the ECDSA algorithm type without the SHA1 digest.

Type of information this allows you to use:

the specified digest algorithm combined with the ECDSA signature algorithm for performing ECDSA signing and verifying as defined in X9.62. The only digest method supported in this API is SHA1.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_DIGEST_SPECIFIER` structure:

```
typedef struct {
  B_INFO_TYPE digestInfoType;
  POINTER digestInfoParams;
} B_DIGEST_SPECIFIER;
```

Crypto-C 4.3 supports only `AI_SHA1` as a *`digestInfoType`*, with `NULL_PTR` as the *`digestInfoParams`*.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `B_DIGEST_SPECIFIER` structure.

Crypto-C procedures to use with algorithm object:

`B_SignInit`, `B_SignUpdate`, `B_SignFinal`, `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal`. You must pass an initialized random algorithm in `B_SignFinal`, but may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all other *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

For signing, `AM_ECFP_DSA_SIGN` for odd prime fields and `AM_ECF2POLY_DSA_SIGN` for an even characteristic. For verifying, `AM_ECFP_DSA_VERIFY` for odd prime fields and

`AM_ECF2POLY_DSA_VERIFY` for even characteristic. Also required is the appropriate AM for the digest specified, for instance, `AM_SHA` when the *`digestInfoType`* is `AI_SHA1`.

Key info types for keyObject in B_SignInit:

`KI_ECPrivate` or `KI_ECPrivateBER`.

Key info types for keyObject in B_VerifyInit:

`KI_ECPublic` or `KI_ECPublicBER`.

Output considerations:

The *`signature`* result of `B_SignFinal` is a BER-encoded value of type SEQUENCE (INTEGER, INTEGER) where the first field is the value *r* and the second field is the value *s*, as defined in section 5.3.1 of X9.57 Draft. The size of *`signature`* is 6 + (2 · (length of order)) bytes. For instance, if the order is 160 bits (20 bytes), the *`signature`* will be 46 bytes long.

AI_EC_ES

Purpose:

This AI allows you to perform public-key encryption or private-key decryption using the Elliptic-Curve Authenticated Encryption System, where ciphertext includes the SHA1 digest as well as encrypted plaintext.

Type of information this allows you to use:

the elliptic curve authenticated encryption scheme as defined in X9.63 Draft, as of 10/97.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You must pass an initialized random algorithm in `B_EncryptFinal`, but may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all other *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_ECFP_ENCRYPT` for encryption and `AM_ECFP_DECRYPT` for decryption with odd prime fields, `AM_ECF2POLY_ENCRYPT` for encryption and `AM_ECF2POLY_DECRYPT` for decryption with even characteristic.

Output Considerations:

The encrypted data can be as much as ((21 + 2 · (the size of a field element in bytes) + (length of input in bytes)) bytes long.

AI_ECKeyGen

Purpose:

This AI allows you to generate an elliptic curve key pair for given EC parameters.

Type of information this allows you to use:

the parameters for generating a compatible elliptic curve key pair. The precomputed table values from `AI_ECAcceleratorTable` can optionally be used to accelerate this operation.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_EC_PARAMS` structure:

```
typedef struct {
  B_INFO_TYPE parameterInfoType;        /* used to interpret EC parameters */
  POINTER     parameterInfoValue;              /* describes elliptic curve */
} B_EC_PARAMS;
```

where *`parameterInfoType`* must be `AI_ECParameters` or `AI_ECParametersBER`, and *`parameterInfoValue`* must be a pointer to an `A_EC_PARAMS` structure or to an `ITEM` containing the BER-encoded EC Parameters.:

```
typedef struct {
  unsigned int version;                          /* implementation version */
  unsigned int fieldType;                     /* indicates type of base field */
  ITEM         fieldInfo;       /* The prime number if fieldType = FT_FP; */
                 /* the basis polynomial if fieldType = FT_F2_POLYNOMIAL; */
                  /* and the degree of the field if fieldType = FT_F2_ONB */
  ITEM         coeffA;                       /* elliptic curve coefficient */
  ITEM         coeffB;                       /* elliptic curve coefficient */
  ITEM         base;                    /* elliptic curve group generator */
  ITEM         order;         /* order of subgroup's generating element */
  ITEM         cofactor;                   /* the cofactor of the subgroup */
  unsigned int pointRepresentation;        /* not used. */
                                    /* set to CI_NO_COMPRESS as a default */
  unsigned int fieldElementBits;             /* field element size in bits */
} A_EC_PARAMS;
```

Format of *info* returned by B_GetAlgorithmInfo:

`B_GetAlgorithmInfo` is not supported with this AI. If called, it will return an error.

Crypto-C procedures to use with algorithm object:

`B_GenerateInit` and `B_GenerateKeypair`.

`B_GenerateKeypair` sets the *`publicKey`* key object with the `KI_ECPublic` or `KI_ECPublicBER` information; and it sets the *`privateKey`* key object with the `KI_ECPrivate` or `KI_ECPrivateBER` information. You must pass an initialized random algorithm to `B_GenerateKeypair`.

Algorithm methods to include in application's algorithm chooser:

`AM_ECFP_KEY_GEN` for odd prime fields and `AM_ECF2POLY_KEY_GEN` for even characteristic.

AI_ECParameters

Purpose:

This AI allows you to specify EC parameters to be used for elliptic curve operations.

Type of information this allows you to use:

information that was either generated by `AI_ECParamGen` or which has been imported from an external source. An algorithm object of type `AI_ECParameters` is initialized either as the *`resultAlgorithmObject`* from a call to `B_GenerateParameters` (`AI_ECParamGen`) or by simply calling `B_SetAlgorithmInfo` (`AI_ECParameters` or `AI_ECParameters_BER`).

The resulting `AI_ECParameters` algorithm object may then be used with `AI_ECKeyGen` to generate keys or with `AI_EC_DHKeyAgree` to execute key agreements.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `A_EC_PARAMS` structure:

```
typedef struct {
  unsigned int version;                          /* implementation version */
  unsigned int fieldType;                   /* indicates type of base field */
  ITEM         fieldInfo;       /* The prime number if fieldType = FT_FP; */
                  /* the basis polynomial if fieldType = FT_F2_POLYNOMIAL; */
                   /* and the degree of the field if fieldType = FT_F2_ONB */
  ITEM         coeffA;                        /* elliptic curve coefficient */
  ITEM         coeffB;                        /* elliptic curve coefficient */
  ITEM         base;                     /* base point of elliptic curve */
  ITEM         order;           /* order of subgroup's generating element */
  ITEM         cofactor;                    /* the cofactor of the subgroup */
  unsigned int pointRepresentation;         /* not used. */
                                      /* set to CI_NO_COMPRESS as a default */
  unsigned int fieldElementBits;              /* field element size in bits */
} A_EC_PARAMS;
```

Set the arguments as follows:

Argument	Values	Comments
version	`1`	Represents the current ANSI X9.62 version
fieldType	`FT_FP`	Odd prime field
	`FT_F2_ONB`	Even characteristic, optimal normal basis
	`FT_F2_POLYNOMIAL`	Even characteristic, polynomial basis
	`FT_X962_NAMED_CURVE_TYPE`	ANSI X9.62 Named Curve Identifier. The possible values are listed in the next table. Keys will be generated that conform to the named curve's definition per ANSI X9.62. The NAMED CURVE representation will be used in all resulting BER public key encodings. If this option is used, all of the following parameters in this data structure will be ignored and therefore do not need to be initialized.
coeffA	`big-endian string of OCTETS`	The elliptic curve's "A" coefficient
coeffB	`big-endian string of OCTETS`	The elliptic curve's "B" coefficient
base	`big-endian string of OCTETS`	The base point or "group generator"
order	`big-endian string of OCTETS`	The order of the subgroup's generating element
cofactor	`big-endian string of OCTETS`	The cofactor of the subgroup
pointRepresentation	`CI_NO_COMPRESS`	Not used. Set to CI_NO_COMPRESS as a default.
fieldElementBits	`Integer`	The field element size in bits

The possible values for `FT_X962_NAMED_CURVE_TYPE` are:

FT_X962_NAMED_CURVE_TYPE	X9.62 Annex J Reference
FT_C2PNB163V1	4.1 Example 1
FT_C2PNB163V2	4.1 Example 2
FT_C2PNB163V3	4.1 Example 3
FT_C2PNB176W1	4.2 Example 1
FT_C2TNB191V1	4.3 Example 1
FT_C2TNB191V2	4.3 Example 2
FT_C2TNB191V3	4.3 Example 3
FT_C2ONB191V4	4.3 Example 4
FT_C2ONB191V5	4.3 Example 5
FT_C2PNB208W1	4.4 Example 1
FT_C2TNB239V1	4.5 Example 1
FT_C2TNB239V2	4.5 Example 2
FT_C2TNB239V3	4.5 Example 3
FT_C2ONB239V4	4.5 Example 4
FT_C2ONB239V5	4.5 Example 5
FT_C2PNB272W1	4.6 Example 1
FT_C2PNB304W1	4.7 Example 1
FT_C2TNB359V1	4.8 Example 1
FT_C2PNB368W1	4.9 Example 1
FT_C2TNB431R1	4.10 Example 1
FT_PRIME192V1	5.1 Example 1
FT_PRIME192V2	5.1 Example 2
FT_PRIME192V3	5.1 Example 3
FT_PRIME239V1	5.2 Example 1
FT_PRIME239V2	5.2 Example 2
FT_PRIME239V3	5.2 Example 3
FT_PRIME256V1	5.3 Example 1

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `A_EC_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_SetAlgorithmInfo` and `B_GetAlgorithmInfo`.

Algorithm methods to include in application's algorithm chooser:

None.

Compatible representation:

`AI_ECParametersBER`

Note: ANSI X9.62 "NAMED CURVE" representation is not preserved when calling `B_GetAlgorithmInfo` in `AI_ECParameters`. NAMED CURVE representation can be asserted when calling `B_SetAlgorithmInfo` by inserting an instance of `FT_X962_NAMED_CURVE_TYPE` into the argument *`fieldType`*. NAMED CURVE representation is propagated into the BER encoding.

AI_ECParametersBER

Purpose:

This AI allows you to specify EC parameters to be used for elliptic curve operations.

Type of information this allows you to use:

the encoding of an ANSI X9.62 Version 1 algorithm identifier that specifies an elliptic curve. An equivalent set of parameters to those produced by `AI_ECParamGen` is decoded and stored in the resulting algorithm object.

The resulting `AI_ECParameters` algorithm object may then be used with `AI_ECKeyGen` to generate keys or with `AI_EC_DHKeyAgree` to execute key agreements.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that contains the ANSI X9.62-compliant BER encoding of an elliptic curve's parameters. In compliance with X9.62, you can specify the `CHOICE` of either a full EC CURVE definition or a NAMED CURVE definition. Both ANSI X9.62 uncompressed and hybrid base points are decoded.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `ITEM` structure that contains the ANSI X9.62-compliant BER encoding of an elliptic curve's parameters.

Crypto-C procedures to use with algorithm object:

`B_SetAlgorithmInfo` and `B_GetAlgorithmInfo`.

Algorithm methods to include in application's algorithm chooser:

None.

Compatible representation:

`AI_ECParameters.`

Note: The X9.62 "NAMED CURVE" representation is not preserved when calling `B_GetAlgorithmInfo` in `AI_ECParameters`. NAMED CURVE representation can be asserted when calling `B_SetAlgorithmInfo` in `AI_ECParameters`; however, `B_GetAlgorithmInfo` always obtains the full description of the elliptic curve's parameters.

AI_ECParamGen

Purpose:

This AI allows you to generate EC parameters to be used for elliptic curve operations.

Type of information this allows you to use:

the input parameters for generating an elliptic curve system as defined in X9.62 and IEEE P1363 Draft.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a B_EC_PARAM_GEN_PARAMS structure:

```
typedef struct {
  unsigned int version;                           /* implementation version */
  unsigned int fieldType;              /* base field for the elliptic curve */
  unsigned int fieldElementBits;         /* length of field element in bits */
  unsigned int compressIndicator;                         /* ignored for now */
  unsigned int minOrderBits;    /* minimum size of group generated by base */
                            /* input of 0 defaults to fieldElementBits - 7 */
  unsigned int trialDivBound;      /* maximum size of second largest prime */
                                  /*  subgroup of group generated by base  */
                                           /*  input of 0 defaults to 255  */
  unsigned int tableLookup;            /* characteristic 2 only. Set if the */
                                  /*  use of precomputed params is desired */
} B_EC_PARAM_GEN_PARAMS;
```

Set the arguments as follows:

Argument	Values	Comments
version	0	Only value currently available; allows growth for future versions
fieldType	FT_FP	odd prime field
	FT_F2_ONB	even characteristic, optimal normal basis
	FT_F2_POLYNOMIAL	even characteristic, polynomial basis
compressIndicator	CI_NO_COMPRESS	do not compress the base or public key

Argument	Values	Comments
fieldElementBits	64 - 384 bits	*fieldType* = FT_FP or FT_F2_POLYNOMIAL
	65, 66, 69, 74, 81, 82, 83, 86, 89, 90, 95, 98, 99, 100, 105, 106, 113, 119, 130, 131, 134, 135, 138, 146, 148, 155, 158, 162, 172, 173, 174, 178, 179, 180, 183, 186, 189, 191, 194, 196, 209, 210, 221, 226, 230, 231, 233, 239, 243, 245, 251, 254, 261, 268, 270, 273, 278, 281, 292, 293, 299, 303, 306, 309, 316, 323, 326, 329, 330, 338, 346, 348, 350, 354, 359, 371, 372, 375, 378	*fieldType* = FT_F2_ONB
minOrderBits	0 (recommended); 1 to *fieldElementBits*	0 tells Crypto-C to choose the value. Note that not all values in the range 1 - *fieldElementBits* are secure. Must be set to 0 if *tableLookup* = 1.
trialDivBound	0 (recommended); 1 - 384	0 tells Crypto-C to choose the value. Must be set to 0 if *tableLookup* = 1.
tableLookup	0 or 1	set to 0 if *fieldType* = FT_FP set to 0 if *fieldType* = FT_F2_ONB or FT_F2_POLYNOMIAL, and you want Crypto-C to generate new parameters from scratch. *minOrderBits* and *trialDivBound* may be non-zero. set to 1 if *fieldType* = FT_F2_ONB or FT_F2_POLYNOMIAL, and you want to generate curves using table lookup. Curve generation will be fast, but *minOrderBits* and *trialDivBound* must be set to 0.

Note: The parameter range given above for *minOrderBits* includes values that are not secure. If you pass 0 for *minOrderBits,* Crypto-C will choose the value for you. You should only pass a non-zero value if you are certain that you are fully aware of the underlying cryptographic issues.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a B_EC_PARAM_GEN_PARAMS structure (see above).

Crypto-C procedures to use with algorithm object:

`B_GenerateInit` and `B_GenerateParameters`. `B_GenerateParameters` sets the *`resultAlgorithmObject`* with the parameter information. You must pass an initialized random algorithm to `B_GenerateParameters`.

Algorithm methods to include in application's algorithm chooser:

`AM_ECFP_PARAM_GEN` for odd prime fields and `AM_ECF2POLY_PARAM_GEN` for even characteristic.

Notes:

Generating an elliptic curve for even characteristic without table lookup (*`fieldType`* = `FT_F2_ONB` or `FT_F2_POLYNOMIAL` and *`tableLookup`* = 0) can be extremely time-consuming, taking several hours in some cases. In general, larger values for *`minOrderBits`* mean longer times for curve generation. Therefore, if you wish to generate curves for even characteristic, but do not want to use table lookup, you can speed curve generation by setting a smaller value for *`minOrderBits`*. Remember, however, that the size of *`minOrderBits`* is directly tied to the security of your elliptic curve cryptosystem. Setting *`minOrderBits`* allows you to make a trade-off between the time it takes to generate curves and the security of your system.

AI_ECPubKey

Purpose:

This AI allows you to specify an EC public key and underlying EC parameters in order to build an acceleration table.

Type of information this allows you to use:

the encoding of an ANSI X9.62 Version 1 algorithm identifier that specifies an elliptic curve public key. An initialized `AI_ECPubKeyBER` or `AI_ECPubKey` algorithm object can be used as the input parameter *`ParameterInfoType`* when initializing an `AI_ECBuildPubKeyAccelTable`.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `A_EC_PUBLIC_KEY` structure:

```
typedef struct {
  A_EC_PARAMS curveParams;     /* the underlying elliptic curve parameters */
  ITEM publicKey;                                     /* public component */
} A_EC_PUBLIC_KEY;

typedef struct {
  unsigned int version;                         /* implementation version */
  unsigned int fieldType;                 /* indicates type of base field */
  ITEM         fieldInfo;       /* The prime number if fieldType = FT_FP; */
                  /* the basis polynomial if fieldType = FT_F2_POLYNOMIAL; */
                   /* and the degree of the field if fieldType = FT_F2_ONB */
  ITEM         coeffA;                       /* elliptic curve coefficient */
  ITEM         coeffB;                       /* elliptic curve coefficient */
  ITEM         base;                    /* elliptic curve group generator */
  ITEM         order;           /* order of subgroup's generating element */
  ITEM         cofactor;                  /* the cofactor of the subgroup */
  unsigned int pointRepresentation;         /*not used */
                                     /* set to CI_NO_COMPRESS as a default */
  unsigned int fieldElementBits;            /* field element size in bits */
} A_EC_PARAMS;
```

Each `ITEM` supplies an integer in canonical format, where the `ITEM`'s *`data`* points to an unsigned byte array -- most significant byte first -- and the `ITEM`'s *`len`* gives its length. For all `ITEM` values except the public component (x) and the curve parameter *`base`*, leading zeros are stripped before they are copied to the key object. `A_EC_PUBLIC_KEY` can be initialized by calling `B_GetKeyInfo` from a key object of type `KI_ECPublic`.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `A_EC_PUBLIC_KEY` structure.

Crypto-C procedures to use with algorithm object:

`B_SetAlgorithmInfo` and `B_GetAlgorithmInfo`.

Can get this info type if algorithm object already has:

`AI_ECPubKey` or `AI_ECPubKeyBER`.

AI_ECPubKeyBER

Purpose:

This AI allows you to specify an EC public key and underlying EC parameters in order to build an acceleration table.

Type of information this allows you to use:

the encoding of an ANSI X9.62 Version 1 algorithm identifier that specifies an elliptic curve public key. An initialized `AI_ECPubKeyBER` or `AI_ECPubKey` algorithm object can be used as the input parameter *`ParameterInfoType`* when initializing an `AI_ECBuildPubKeyAccelTable`.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that contains the ANSI X9.62-compliant BER encoding of a public key and its elliptic curve parameters. In compliance with X9.62, you can specify the `CHOICE` of either a full EC CURVE definition or a NAMED CURVE definition. Both ANSI X9.62 uncompressed and hybrid base points are decoded.

The BER encoding can be obtained by calling `B_GetKeyInfo` from a key object of type `KI_ECPublicBER`.

The encoding is converted to DER before it is copied to the key object. `B_SetKeyInfo` returns `BE_WRONG_KEY_INFO` if the BER encoding is incorrect.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that contains the address and length of the BER encoding.

Crypto-C procedures to use with algorithm object:

`B_SetAlgorithmInfo` and `B_GetAlgorithmInfo`.

Can get this info type if algorithm object already has:

`AI_ECPubKey` or `AI_ECPubKeyBER`.

AI_FeedbackCipher

Purpose:

This AI allows you to perform various kinds of block cipher encryption or decryption with feedback. The parameters of this AI include encryption method, feedback method, and padding method.

Type of information this allows you to use:

a descriptor for block ciphers with feedback, as defined in X9.52.

Format of *info* supplied to B_SetAlgorithmInfo:

a pointer to a B_BLK_CIPHER_W_FEEDBACK_PARAMS structure:

```
typedef struct {
  unsigned char *encryptionMethodName;             /* examples include */
                                                    /* “des”, “des_ede” */
  POINTER        encryptionParams;            /* e.g., RC5 parameters */
  unsigned char *feedbackMethodName;          /*feedback method name, */
                                                        /* e.g., “cbc” */
  POINTER        feedbackParams;        /* Points at init vector ITEM */
                                 /* for all feedback modes except cfb */
  unsigned char *paddingMethodName;           /* padding method name, */
                                                        /* e.g., “pad” */
  POINTER        paddingParams;    /* Ignored for now, but may be used */
                                           /* for new padding schemes */
} B_BLK_CIPHER_W_FEEDBACK_PARAMS;
```

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a structure of type B_BLK_CIPHER_W_FEEDBACK_PARAMS.

Type of padding schemes available:

Padding Scheme	Reference String	Comments
Standard CBC padding	`"pad"`	
No padding	`"nopad"`	Input length must be a multiple of cipher block length (= 8 for all but rc5_64)
Stream	`"stream"`	Only available when using CFB in 1-bit and 8-bit modes.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

the block cipher AM and the feedback mode AM specified by *`encryptionMethodName`* and *`feedbackMethodName`*, respectively.

Table 2-1 **Algorithm methods for block ciphers**

encryptionMethodName	Algorithm methods to include in chooser	Parameters	Key Size
Encryption			
`"des"`	`AM_DES_ENCRYPT;`	null	8 bytes
`"des_ede"`	`AM_DES_EDE_ENCRYPT;`	null	24 bytes
`"desx"`	`AM_DESX_ENCRYPT;`	null	24 bytes
`"rc2"`	`AM_RC2_ENCRYPT;`	Pointer to an A_RC2_PARAMS structure	1 - 128 bytes
`"rc5"`	`AM_RC5_ENCRYPT;`	Pointer to an A_RC5_PARAMS structure	0 - 255 bytes; 32-bit word
`"rc5_64"`	`AM_RC5_64ENCRYPT;`	Pointer to an A_RC5_PARAMS structure	0 - 255 bytes; 64-bit word
`"rc6"`	`AM_RC6_ENCRYPT;`	Pointer to an A_RC6_PARAMS structure	

Table 2-1 **Algorithm methods for block ciphers**

encryptionMethodName	Algorithm methods to include in chooser	Parameters	Key Size
Decryption			
“des”	AM_DES_DECRYPT;	null	8 bytes
“des_ede”	AM_DES_EDE_DECRYPT;	null	24 bytes
“desx”	AM_DESX_DECRYPT;	null	24 bytes
“rc2”	AM_RC2_DECRYPT;	Pointer to an A_RC2_PARAMS structure	1 - 128 bytes
“rc5”	AM_RC5_DECRYPT;	Pointer to an A_RC5_PARAMS structure	0 - 255 bytes; 32-bit word
“rc5_64”	AM_RC5_64DECRYPT;	Pointer to an A_RC5_PARAMS structure	0 - 255 bytes; 64-bit word
"rc6"	AM_RC6_DECRYPT;	Pointer to an A_RC6_PARAMS structure	

The RC2 algorithm methods, AM_RC2_ENCRYPT and AM_RC2_DECRYPT, require an A_RC2_PARAMS structure:

```
typedef struct {
  unsigned int effectiveKeyBits;            /* effective key size in bits */
} A_RC2_PARAMS;
```

The RC5 algorithm methods, AM_RC5_ENCRYPT, AM_RC564_ENCRYPT, AM_RC5_DECRYPT, and AM_RC564_DECRYPT, require an A_RC5_PARAMS structure. Note that the 64-bit versions of the RC5 algorithm, AM_RC5_64_ENCRYPT and AM_RC5_64_DECRYPT, are evaluation implementations and are not optimized in Crypto-C 4.3:

```
typedef struct {
  unsigned int version;   / * currently version 1.0 -- defined as 0x10 */
  unsigned int rounds;                   /* number of rounds (0 - 255) */
  unsigned int wordSizeInBits;      /* 32 for “rc5” or 64 for “rc5_64” */
} A_RC5_PARAMS;
```

The RC6 algorithm methods, AM_RC6_ENCRYPT and AM_RC6_DECRYPT, require an

A_RC6_PARAMS structure:

```
typedef struct {
  unsigned int rounds;                           /* number of rounds (20) */
} A_RC6_PARAMS;
```

Note: The A_RC6_PARAMS structure requires a parameter specifying the number of rounds. Currently, Crypto-C accepts only 20. Future versions may accept other round counts.

Table 2-2 **Algorithm methods for feedback modes**

feedbackMethodName	Algorithm methods to include in chooser	Parameter Type
Encryption		
"cbc"	AM_CBC_ENCRYPT;	ITEM that contains the initialization vector
"cbc_interleaved"	AM_CBC_INTER_LEAVED_ENCRYPT;	ITEM that contains the initialization vector
"cfb"	AM_CFB_ENCRYPT;	B_CFB_PARAMS
"cfb_pipelined"	AM_CFB_PIPELINED_ENCRYPT;	B_CFB_PARAMS
"ecb"	AM_ECB_ENCRYPT;	unsigned int that gives the block length
"ofb"	AM_OFB_ENCRYPT;	ITEM that contains the initialization vector
"ofb_pipelined"	AM_OFB_PIPELINED_ENCRYPT;	ITEM that contains the initialization vector
Decryption		
"cbc"	AM_CBC_DECRYPT;	ITEM that contains the initialization vector
"cbc_interleaved"	AM_CBC_INTER_LEAVED_DECRYPT;	ITEM that contains the initialization vector
"cfb"	AM_CFB_DECRYPT;	B_CFB_PARAMS
"cfb_pipelined"	AM_CFB_PIPELINED_DECRYPT;	B_CFB_PARAMS
"ecb"	AM_ECB_DECRYPT;	unsigned int that gives the block length

Table 2-2 **Algorithm methods for feedback modes**

feedbackMethodName	**Algorithm methods to include in chooser**	**Parameter Type**
"ofb"	AM_OFB_DECRYPT;	ITEM that contains the initialization vector
"ofb_pipelined"	AM_OFB_PIPELINED_DECRYPT;	ITEM that contains the initialization vector

The CFB modes require a B_CFB_PARAMS structure:

```
typedef struct {
  ITEM ivItem;
  unsigned int transferSize;
} B_CFB_PARAMS;
```

Note: The initialization vector should be the same size as the block. In particular, the IV should be 8 bytes, except for the RC5 algorithm implemented with a 64-bit word, which requires a 16-byte IV, or the RC6 algorithm.

Key info types for keyObject in B_EncryptInit or B_DecryptInit:

Depends on cipher type, as follows:

Cipher	**KIs**
DES	KI_Item, KI_DES8, KI_DES8Strong, KI_8Byte
Triple DES	KI_Item, KI_DES24Strong, KI_24Byte
DESX	KI_Item, KI_DES24Strong, KI_24Byte
RC2	KI_Item, KI_8Byte
RC5	KI_Item, which gives the address and length of the RC5 Key.
RC6	KI_Item

Compatible representations:

Cipher	Compatible representations
DES with CBC mode	`AI_DES_CBC_IV8`, `AI_DES_CBCPadIV8`, `AI_DES_CBCPadBER`, `AI_DES_CBCPadPEM`, and `AI_DES_CBC_BSAFE1`
TDES with CBC mode	`AI_DES_EDE3_CBC_IV8`, `AI_DES_EDE3_CBCPadIV8`, and `AI_DES_EDE3_CBCPadBER`
DESX with CBC mode	`AI_DESX_CBC_IV8`, `AI_DESX_CBCPadIV8`, `AI_DESX_CBCPadBER`, and `AI_DESX_CBC_BSAFE1`
RC2 with CBC mode	`AI_RC2_CBC`, `AI_RC2_CBCPad`, `AI_RC2_CBCPadBER`, and `AI_RC2_CBCPadPEM`
RC5 with CBC mode and 32-bit word	`AI_RC5_CBC`, and `AI_RC5_CBCPad`
RC6 with CBC mode	`AI_RC6_CBC and AI_RC6_CBCPad`

Output considerations:

For encryption, when padding is used, the total number of output bytes can be as many as one block size more than the total input. For decryption, the output buffer should be the same size as the input buffer, even if padding was used.

AI_HMAC

Purpose:

This AI allows you to create a message authentication code using a keyed hashing algorithm called Hash-Based Message Authentication Code (HMAC).

Type of information this allows you to use:

the specified digest algorithm for performing HMAC as defined in the SET draft standard, a subset of the Draft-IETF-IPSEC-AH-HMAC-SHA-04.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_DIGEST_SPECIFIER` structure:

```
typedef struct {
  B_INFO_TYPE digestInfoType;
  POINTER digestInfoParams;
} B_DIGEST_SPECIFIER;
```

Crypto-C 4.3 supports `AI_SHA1` and `AI_MD5` as a *`digestInfoType`*, with `NULL_PTR` as the *`digestInfoParams`*.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `B_DIGEST_SPECIFIER` structure.

Crypto-C procedures to use with algorithm object:

`B_DigestInit`, `B_DigestUpdate`, and `B_DigestFinal`. You must pass a key object compatible with `KI_Item` with at least one byte of keying material as the *`keyObject`* argument in `B_DigestInit`. It is recommended that minimally 20 bytes of key be used when SHA1 is the hash function. More bytes are recommended if the key bytes are weakly random (low entropy keys).

Algorithm methods to include in application's algorithm chooser:

The appropriate AM for the digest specified; for instance, `AM_SHA` when the *`digestInfoType`* is `AI_SHA1`, and `AM_MD5` for `AI_MD5`.

Output considerations:

The output of `B_DigestFinal` will be the length of the output of the digest specified; for instance, 20 bytes when *`digestInfoType`* is `AI_SHA1`.

AI_HW_Random

Purpose:

This AI allows you to generate random bytes using a hardware device.

Type of information this allows you to use:

random bytes generated by your hardware device.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of info returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_RandomInit` and `B_GenerateRandomBytes`, and as the *`randomAlgorithm`* argument to other procedures.

Algorithm methods to include in application's algorithm chooser:

`AM_HW_RANDOM`.

Notes:

Can only be used in conjunction with a hardware implementation. `AM_HW_RANDOM` can only be used if you have called `B_CreateSessionChooser` for your application. For an example of a hardware implementation using `AM_HW_RANDOM`, see the *Intel Security Hardware User's Guide*, available on the Crypto-C CD.

AI_KeypairTokenGen

Purpose:

This AI allows you to generate the token form of a public/private key pair with a hardware device.

Type of information this allows you to use:

the parameters for generating the token form of a public/private key pair. The BSAFE Hardware API (BHAPI) supports token forms of RSA strong key pair generation as defined in PKCS #1 and DSA key pair generation as defined in FIPS PUB 186.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an A_KEYPAIR_SPECIFIER structure:

```
typedef struct {
  A_KEYPAIR_DEFINER privateKeyDef;       /* Specifications for private key */
  A_KEYPAIR_DEFINER publicKeyDef;         /* Specifications for public key */
  POINTER           keyParams;  /* Points to RSA params in RSA case, i.e., */
                                              /* A_RSA_KEY_GEN_PARAMS. */
                                    /* Points to DSA params in DSA case. */
  unsigned char    *cipherName;       /* String tag for key's cipher class */
                                          /* Either “rsa” or “dsa” to tag */
} A_KEYPAIR_SPECIFIER;
```

where A_KEYPAIR_DEFINER is defined by:

```
typedef struct {
  unsigned int keyUsage;                          /* X509 key usage bit map */
  UINT4        lifeTime;            /* Key lifetime; under consideration */
  unsigned int protectFlag;                /* Store key in encrypted form */
} A_KEYPAIR_DEFINER;
```

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an A_KEYPAIR_SPECIFIER structure (see above).

Crypto-C procedures to use with algorithm object:

B_GenerateInit and B_GenerateKeypair. If hardware is present, B_GenerateKeypair sets the *publicKeyDef* and *privateKeyDef* key objects with the public and private key information from KI_Token. If no hardware is present, and software emulation methods have been included in the hardware chooser, B_GenerateKeypair sets the *publicKeyDef* and *privateKeyDef* key objects with the public and private key information from KI_KeypairToken. You must pass an initialized random algorithm to B_GenerateKeypair, unless the hardware manufacturer has it internally implemented. In this case, a properly cast NULL_PTR should be used.

Algorithm methods to include in application's algorithm chooser:

the key-pair generation AM specified by *cipherName*:

cipherName	**Algorithm methods to include in chooser**
"dsa"	AM_DSA_KEY_TOKEN_GEN
"rsa"	AM_RSA_KEY_TOKEN_GEN

Notes:

Can only be used in conjunction with a hardware implementation or software emulation. AI_KeypairTokenGen can only be used if you have called B_CreateSessionChooser for your application.

The corresponding software-emulation methods passed to B_CreateSessionChooser via the HARDWARE_CHOOSER list are a HW_TABLE_ENTRY SF_RSA_KEY_TOKEN_GEN for RSA keys and a HW_TABLE_ENTRY SF_DSA_KEY_TOKEN_GEN for DSA keys. These provide software support if hardware is unavailable. These methods can be utilized only by including the software chooser inside the hardware chooser table.

At B_GenerateInit the key generation object is bound to the hardware device, if one is available. If no hardware device is present, the key-generation object is bound to the software-emulation method if it has been included in the hardware chooser; it defaults to the null method otherwise. For example, for an RSA key token, if no hardware is present, the key generation object is bound to SF_RSA_KEY_TOKEN_GEN if it is included in the hardware chooser and defaults to AM_RSA_KEY_TOKEN_GEN otherwise.

AI_MAC

Purpose:

This AI allows you to create a variable-length message authentication code (MAC) on a non-linear feedback shift register.

Type of information this allows you to use:

the MAC length parameter for the MAC algorithm

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_MAC_PARAMS` structure:

```
typedef struct {
  unsigned int macLen;                              /* length of MAC value */
} B_MAC_PARAMS;
```

The minimum *`macLen`* is 2.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `B_MAC_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_DigestInit`, `B_DigestUpdate`, and `B_DigestFinal`. Supply `NULL_PTR` for the *`keyObject`* argument in `B_DigestInit`.

Algorithm methods to include in application's algorithm chooser:

`AM_MAC`.

Output considerations:

The output of `B_DigestFinal` will be *`macLen`* bytes long.

AI_MD

Purpose:

Deprecated. This AI is included only for backward compatibility.

Type of information this allows you to use:

the message digest (MD) algorithm as defined by BSAFE 1.x.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_DigestInit`, `B_DigestUpdate`, and `B_DigestFinal`. Supply `NULL_PTR` for the *`keyObject`* argument in `B_DigestInit`.

Algorithm methods to include in application's algorithm chooser:

`AM_MD`.

Output considerations:

The output of `B_DigestFinal` will be 16 bytes long.

AI_MD2

Purpose:

This AI allows you to create a message digest using the MD2 digest algorithm as defined in RFC 1319. This algorithm processes input data 16 bytes at a time, but the length of the input does not have to be a multiple of 16 as the algorithm pads automatically. The primary use for this AI is to authenticate data. Other algorithms that can be used for message digesting are `AI_MD5` and `AI_SHA1` and their variants. See `AI_MD2_BER` for the MD2 algorithm type with BER encoding. See `AI_MD2_PEM` for the MD2 algorithm type with PEM encoding.

Type of information this allows you to use:

the MD2 message digest algorithm as defined in RFC 1319.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_DigestInit`, `B_DigestUpdate`, and `B_DigestFinal`. Supply `NULL_PTR` for the *`keyObject`* argument in `B_DigestInit`.

Algorithm methods to include in application's algorithm chooser:

`AM_MD2`.

Compatible representation:

`AI_MD2_BER`, `AI_MD2_PEM`.

Output considerations:

The output of `B_DigestFinal` will be 16 bytes long.

AI_MD2_BER

Purpose:

This AI is similar to `AI_MD2` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_MD2`, `AI_MD2_BER` or `AI_MD2_PEM`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 2, 2`". Also see `AI_MD2.`

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the MD2 message digest algorithm as defined in RFC 1319.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies a message digest algorithm other than MD2.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_DigestInit`, `B_DigestUpdate`, and `B_DigestFinal`. Supply `NULL_PTR` for the *`keyObject`* argument in `B_DigestInit`.

Algorithm methods to include in application's algorithm chooser:

`AM_MD2`.

Compatible representation:

`AI_MD2`, `AI_MD2_PEM`.

Output considerations:

The output of `B_DigestFinal` will be 16 bytes long.

AI_MD2_PEM

Purpose:

This AI is similar to AI_MD2 except that it uses the Privacy Enhanced Mail (PEM) format. This AI allows you to parse and create PEM algorithm identifiers such as used in the PEM protocol. You call B_SetAlgorithmInfo to initialize an algorithm object from the encoded algorithm identifier. You call B_GetAlgorithmInfo with this AI to create an encoded algorithm identifier from an algorithm object that was created using AI_MD2, AI_MD2_BER, or AI_MD2_PEM. Also see AI_MD2.

Type of information this allows you to use:

an RFC 1423 identifier that specifies the MD2 message digest algorithm as defined in RFC 1319. This algorithm info type is intended to process the digest identifier in a MIC-Info field in a PEM-encapsulated header.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a null-terminated string (char *) that gives the RSA-MD2 identifier, for example, "RSA-MD2." Space and tab characters are removed from the string before it is copied to the algorithm object. B_SetAlgorithmInfo returns BE_WRONG_ALGORITHM_INFO if the algorithm identifier specifies an identifier other than RSA-MD2.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a null-terminated string that gives the RSA-MD2 identifier.

Crypto-C procedures to use with algorithm object:

B_DigestInit, B_DigestUpdate, and B_DigestFinal. Supply NULL_PTR for the *keyObject* argument in B_DigestInit.

Algorithm methods to include in application's algorithm chooser:

AM_MD2.

Compatible representation:

`AI_MD2`, `AI_MD2_BER`.

Output considerations:

The output of `B_DigestFinal` will be 16 bytes long.

AI_MD2Random

Purpose:

This AI allows you to generate a stream of pseudo-random numbers which are guaranteed to have a very high degree of randomness. Random numbers are used in deriving public and private keys, initialization vectors, etc. This algorithm is the same as `AI_MD5Random` described in RSA Labs Bulletin #8 except that the underlying digest algorithm used is MD2 instead of MD5. The details of the `AI_MD5Random` algorithm are available online from RSA Laboratories at `http://www.rsasecurity.com/rsalabs/bulletins/`.

Other algorithms that can be used to generate pseudo-random numbers are `AI_MD5Random` and `AI_X962Random_V0`.

Type of information this allows you to use:

the MD2-Random algorithm for generating pseudo-random numbers, as defined by RSA Security Inc.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_RandomInit`, `B_RandomUpdate`, and `B_GenerateRandomBytes`, and as the *`randomAlgorithm`* argument to other procedures.

Algorithm methods to include in application's algorithm chooser:

`AM_MD2_RANDOM`.

AI_MD2WithDES_CBCPad

Purpose:

This AI allows you to perform password-based encryption. This means that the input data will be encrypted with a secret key derived from a password, and it can be successfully decrypted only when the correct password is provided. Although this AI can be used to encrypt arbitrary data, its intended primary use is for encrypting private keys when transferring them from one computer system to another, as described in PKCS #8.

This AI employs DES secret-key encryption in cipher-block chaining (CBC) mode with padding, where the secret key is derived from a password using the MD2 message digest algorithm. The details of this algorithm are contained in PKCS #5. DES is defined in FIPS PUB 81, and CBC mode of DES is defined in FIPS PUB 46-1. RFC 1319 describes MD2. Other algorithms that can be used for password-based encryption are `AI_MD5WithDES_CBCPad`, `AI_MD5WithRC2_CBCPad`, `AI_MD2WithRC2_CBCPad`, and `AI_SHA1WithDES_CBCPad`.

Type of information this allows you to use:

the salt and iteration count for the MD2 With DES-CBC password-based encryption algorithm as defined in PKCS #5.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_PBE_PARAMS` structure:

```
typedef struct {
  unsigned char *salt;                    /* pointer to 8-byte salt value */
  unsigned int  iterationCount;                      /* iteration count */
} B_PBE_PARAMS;
```

RSA Security Inc. recommends a minimum iteration count of 1,000. However, for an additional byte or two of security the iteration should be 2^8 to 2^{16}.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `B_PBE_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_MD2` and `AM_DES_CBC_ENCRYPT` for encryption or `AM_DES_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item` that gives the password.

Compatible representation:

`AI_MD2WithDES_CBCPadBER`.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_MD2WithDES_CBCPadBER

Purpose:

This AI is similar to `AI_MD2WithDES_CBCPad` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier that includes the salt and iteration count. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_MD2WithDES_CBCPad` or `AI_MD2WithDES_CBCPadBER`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 5, 1`". Also see `AI_MD2WithDES_CBCPad`.

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the MD2 With DES-CBC password-based encryption algorithm as defined in PKCS #5.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than MD2 With DES-CBC.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_MD2` and `AM_DES_CBC_ENCRYPT` for encryption or `AM_DES_CBC_DECRYPT` for

decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item` that gives the password.

Compatible representation:

`AI_MD2WithDES_CBCPad`.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_MD2WithRC2_CBCPad

Purpose:

This AI allows you to perform password-based encryption. This means that the input data will be encrypted with a secret key derived from a password, and it can be successfully decrypted only when the correct password is provided. Although this AI can be used to encrypt arbitrary data, its intended primary use is for encrypting private keys when transferring them from one computer system to another, as described in PKCS #8.

This AI employs RC2 block cipher with padding, where the secret key is derived from a password using the MD2 message digest algorithm. MD2 is described in RFC 1319. RC2 is described in RFC 2268. The CBC mode is similar to the one used in RC5-CBC, which can be found in RFC 2040. Other algorithms that can be used for password-based encryption are `AI_MD5WithDES_CBCPad`, `AI_MD5WithRC2_CBCPad`, `AI_MD2WithDES_CBCPad`, and `AI_SHA1WithDES_CBCPad`.

Type of information this allows you to use:

the effective key size, salt, and iteration count for the MD2 With RC2-CBC password-based encryption algorithm, as defined by RSA Security Inc.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_RC2_PBE_PARAMS` structure:

```
typedef struct {
  unsigned int  effectiveKeyBits;          /* effective key size in bits */
  unsigned char *salt;                    /* pointer to 8-byte salt value */
  unsigned int  iterationCount;                       /* iteration count */
} B_RC2_PBE_PARAMS;
```

This algorithm will accept a maximum of 1024 effective key bits for domestic use and 40 effective key bits for export. RSA Security Inc. recommends a minimum iteration count of 1,000. However, for an additional byte or two of security, the iteration should be 2^8 to 2^{16}.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a B_RC2_PBE_PARAMS structure (see above).

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, B_EncryptFinal, B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. You may pass (B_ALGORITHM_OBJ)NULL_PTR for all *randomAlgorithm* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_MD2 and AM_RC2_CBC_ENCRYPT for encryption or AM_RC2_CBC_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_Item that gives the password.

Compatible representation:

AI_MD2WithRC2_CBCPadBER.

Output considerations:

During encryption, this AI pads the output. Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_MD2WithRC2_CBCPadBER

Purpose:

This AI is similar to `AI_MD2WithRC2_CBCPad` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier that includes the effective key size, salt and iteration count. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_MD2WithRC2_CBCPad` or `AI_MD2WithRC2_CBCPadBER`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal, is “`42, 134, 72, 134, 247, 13, 1, 5, 4`”. Also see `AI_MD2WithRC2_CBCPad`.

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the MD2 With RC2-CBC password-based encryption algorithm, as defined by RSA Security Inc.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than MD2 With RC2-CBC.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application’s algorithm chooser:

`AM_MD2` and `AM_RC2_CBC_ENCRYPT` for encryption or `AM_RC2_CBC_DECRYPT` for

decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item` that gives the password.

Compatible representation:

`AI_MD2WithRC2_CBCPad`.

Output considerations:

During encryption, this AI pads the output. Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_MD2WithRSAEncryption

Purpose:

This AI allows you to perform signature operations that involve the MD2 digest algorithm and RSA public key algorithm. The digest of a message is created using the MD2 algorithm and then it is signed using PKCS#1 digital signature algorithm. Other algorithms that can be used for the same purpose are `AI_MD5WithRSAEncryption` and `AI_SHA1WithRSAEncryption`. See `AI_MD2WithRSAEncryptionBER` for the same algorithm type with BER encoding.

Type of information this allows you to use:

the MD2 With RSA Encryption signature algorithm that uses the MD2 digest algorithm and the RSA algorithm to create and verify RSA digital signatures as defined in PKCS #1. Note that in order to perform PKCS #1 digital signatures with a 16-byte digest, the RSA key must be at least 360 bits long.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_SignInit`, `B_SignUpdate`, `B_SignFinal`, `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_MD2`, and `AM_RSA_CRT_ENCRYPT`, `AM_RSA_CRT_ENCRYPT_BLIND`, or `AM_RSA_ENCRYPT`, for signature creation; and `AM_RSA_DECRYPT` for signature verification. `AM_RSA_CRT_ENCRYPT_BLIND` performs blinding to protect against timing attacks, whereas `AM_RSA_CRT_ENCRYPT` does not.

Key info types for *keyObject* in B_SignInit:

KI_RSA_CRT, KI_PKCS_RSAPrivate, KI_PKCS_RSAPrivateBER, KI_RSAPrivate or KI_RSAPrivateBSAFE1. Unless you use KI_RSA_CRT for your KI, you must include AM_RSA_ENCRYPT in your application's algorithm chooser.

Key info types for *keyObject* in B_VerifyInit:

KI_RSAPublic, KI_RSAPublicBER, or KI_RSAPublicBSAFE1.

Compatible representation:

AI_MD2WithRSAEncryptionBER.

Output considerations:

The *signature* result of B_SignFinal will be the same size as the RSA key's modulus.

Notes:

Although the RSA signature operation is called "encryption" and the verification operation is called "decryption", the signer uses the digest and the private key and follows the steps needed to decrypt, while the verifier uses the transmitted digest and the public key and follows the steps needed to encrypt.

AI_MD2WithRSAEncryptionBER

Purpose:

This AI is similar to `AI_MD2WithRSAEncryption` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_MD2WithRSAEncryption` or `AI_MD2WithRSAEncryptionBER`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 1, 2`". Also see `AI_MD2WithRSAEncryption`.

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the MD2 With RSA Encryption signature algorithm that uses the MD2 digest algorithm and the RSA algorithm to create and verify RSA digital signatures as defined in PKCS #1.

Note that in order to perform PKCS #1 digital signatures with a 16-byte digest, the RSA key must be at least 360 bits long.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than MD2 With RSA Encryption.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_SignInit`, `B_SignUpdate`, `B_SignFinal`, `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_MD2, and AM_RSA_CRT_ENCRYPT, AM_RSA_CRT_ENCRYPT_BLIND, or AM_RSA_ENCRYPT, for signature creation; and AM_RSA_DECRYPT for signature verification. AM_RSA_CRT_ENCRYPT_BLIND performs blinding to protect against timing attacks, whereas AM_RSA_CRT_ENCRYPT does not.

Key info types for *keyObject* in B_SignInit:

KI_RSA_CRT, KI_PKCS_RSAPrivate, KI_PKCS_RSAPrivateBER, KI_RSAPrivate or KI_RSAPrivateBSAFE1. Unless you use KI_RSA_CRT for your KI, you must include AM_RSA_ENCRYPT in your application's algorithm chooser.

Key info types for *keyObject* in B_VerifyInit:

KI_RSAPublic, KI_RSAPublicBER, or KI_RSAPublicBSAFE1.

Compatible representation:

AI_MD2WithRSAEncryption.

Output considerations:

The *signature* result of B_SignFinal will be the same size as the RSA key's modulus.

Notes:

Although the RSA signature operation is called "encryption" and the verification operation is called "decryption", the signer uses the digest and the private key and follows the steps needed to decrypt, while the verifier uses the transmitted digest and the public key and follows the steps needed to encrypt.

AI_MD5

Purpose:

This AI allows you to create a message digest using the MD5 digest algorithm as defined in RFC 1321. This algorithm processes input data 64 bytes at a time, but the length of the input does not have to be a multiple of 64 because the algorithm pads automatically.

The primary use for this AI is to authenticate data. Other algorithms that can be used for message digesting are `AI_MD2` and `AI_SHA1` and their variants.

Type of information this allows you to use:

the MD5 message digest algorithm as defined in RFC 1321.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_DigestInit`, `B_DigestUpdate`, and `B_DigestFinal`. Supply `NULL_PTR` for the *`keyObject`* argument in `B_DigestInit`.

Algorithm methods to include in application's algorithm chooser:

`AM_MD5`.

Compatible representation:

`AI_MD5_BER`, `AI_MD5_PEM`.

Output considerations:

The output of `B_DigestFinal` will be 16 bytes long.

AI_MD5_BER

Purpose:

This AI is similar to `AI_MD5` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_MD5`, `AI_MD5_BER`, or `AI_MD5_PEM`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal is "`42, 134, 72, 134, 247, 13, 2, 5`".

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the MD5 message digest algorithm as defined in RFC 1321.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies a message digest algorithm other than MD5.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_DigestInit`, `B_DigestUpdate`, and `B_DigestFinal`. Supply `NULL_PTR` for the *`keyObject`* argument in `B_DigestInit`.

Algorithm methods to include in application's algorithm chooser:

`AM_MD5`.

Compatible representation:

`AI_MD5`, `AI_MD5_PEM`.

Output considerations:

The output of `B_DigestFinal` will be 16 bytes long.

AI_MD5_PEM

Purpose:

This AI is similar to `AI_MD5` except that it uses the Privacy Enhanced Mail (PEM) format. This AI allows you to parse and create PEM algorithm identifiers such as those used in the PEM protocol. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_MD5`, `AI_MD5_BER`, or `AI_MD5_PEM`.

Type of information this allows you to use:

an RFC 1423 identifier that specifies the MD5 message digest algorithm as defined in RFC 1321. This algorithm info type is intended to process the digest identifier in a MIC-Info field in a PEM-encapsulated header.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a null-terminated string (`char *`) that gives the RSA-MD5 identifier. For example, "`Rsa-MD5`". Space and tab characters are removed from the string before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an identifier other than RSA-MD5.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a null-terminated string that gives the RSA-MD5 identifier.

Crypto-C procedures to use with algorithm object:

`B_DigestInit`, `B_DigestUpdate`, and `B_DigestFinal`. Supply `NULL_PTR` for the *`keyObject`* argument in `B_DigestInit`.

Algorithm methods to include in application's algorithm chooser:

`AM_MD5`.

Compatible representation:

`AI_MD5` and `AI_MD5_BER`.

Output considerations:

The output of `B_DigestFinal` will be 16 bytes long.

AI_MD5Random

Purpose:

This AI allows you to generate a stream of pseudo-random numbers which are guaranteed to have a very high degree of randomness. Random numbers are used in deriving public and private keys, initialization vectors, etc. This AI uses MD5 as an underlying hashing function. The details of this algorithm are available from RSA Laboratories' Bulletin #8 or online at `http://www.rsasecurity.com/rsalabs/bulletins/`.

Other algorithms that can be used to generate pseudo-random numbers are `AI_MD2Random` and `AI_X962Random_V0.`

Type of information this allows you to use:

the MD5-Random algorithm for generating pseudo-random numbers.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_RandomInit`, `B_RandomUpdate`, and `B_GenerateRandomBytes`, and as the *`randomAlgorithm`* argument to other procedures.

Algorithm methods to include in application's algorithm chooser:

`AM_MD5_RANDOM`.

AI_MD5WithDES_CBCPad

Purpose:

This AI allows you to perform password-based encryption. This means that the input data will be encrypted with a secret key derived from a password, and it can be successfully decrypted only when the correct password is provided. Although this AI can be used to encrypt arbitrary data, its intended primary use is for encrypting private keys when transferring them from one computer system to another, as described in PKCS #8.

This AI employs DES secret-key encryption in cipher-block chaining (CBC) mode with padding, where the secret key is derived from a password using the MD5 message digest algorithm. The details of this algorithm are contained in PKCS #5. DES is defined in FIPS PUB 81, and CBC mode of DES is defined in FIPS PUB 46-1. RFC 1321 describes MD5.

Other algorithms that can be used for password-based encryption are `AI_MD2WithDES_CBCPad`, `AI_MD2WithRC2_CBCPad`, `AI_MD5WithRC2_CBCPad`, and `AI_SHA1WithDES_CBCPad`.

Type of information this allows you to use:

the salt and iteration count for the MD5 With DES-CBC password-based encryption algorithm as defined in PKCS #5. The salt is concatenated with the password before being digested by MD5, and the iteration count specifies how many times the digest needs to be run. The count of 2 indicates that the result of digesting password-and-salt string needs to be run once more through MD5. The first 8 bytes of the final digest become the secret key for the DES cipher after being adjusted for parity as required by FIPS PUB 81, and the last 8 bytes become the initialization vector.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_PBE_PARAMS` structure:

```
typedef struct {
  unsigned char *salt;                    /* pointer to 8-byte salt value */
  unsigned int iterationCount;                           /* iteration count */
} B_PBE_PARAMS;
```

RSA Security Inc. recommends a minimum iteration count of 1,000. However, for an additional byte or two of security the iteration should be 2^8 to 2^{16}.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `B_PBE_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_MD5` and `AM_DES_CBC_ENCRYPT` for encryption or `AM_DES_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item` that gives the password.

Compatible representation:

`AI_MD5WithDES_CBCPadBER`.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_MD5WithDES_CBCPadBER

Purpose:

This AI is similar to `AI_MD5WithDES_CBCPad` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier which includes ASN.1 encoding of the `B_PBE_PARAMS` structure defined in the description of `AI_MD5WithDES_CBCPad`. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_MD5WithDES_CBCPad` or `AI_MD5WithDES_CBCPad`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal is “`42, 134, 72, 134, 247, 13, 1, 5, 3`”.

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the MD5 With DES-CBC password-based encryption algorithm as defined in PKCS #5.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than MD5 With DES-CBC.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_MD5 and AM_DES_CBC_ENCRYPT for encryption or AM_DES_CBC_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_Item that gives the password.

Compatible representation:

AI_MD5WithDES_CBCPad.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_MD5WithRC2_CBCPad

Purpose:

This AI allows you to perform password-based encryption. This means that the input data will be encrypted with a secret key derived from a password, and it can be successfully decrypted only when the correct password is provided. Although this AI can be used to encrypt arbitrary data, its intended primary use is for encrypting private keys when transferring them from one computer system to another, as described in PKCS #8.

This AI employs the RC2 block cipher with padding, where the secret key is derived from a password using the MD5 message digest algorithm. MD5 is described in RFC 1321. The RC2 cipher is described in RFC 2268. The CBC mode is similar to the one used in RC5-CBC, which can be found in RFC 2040.

Other algorithms that can be used for password-based encryption are `AI_MD2WithDES_CBCPad`, `AI_MD2WithRC2_CBCPad`, `AI_MD5WithDES_CBCPad`, and `AI_SHA1WithDES_CBCPad`.

Type of information this allows you to use:

the effective key size, salt, and iteration count for the MD5 With RC2-CBC password-based encryption algorithm. The salt is concatenated with the password before being processed by MD5, and the iteration count specifies how many times the digest needs to be run. The count of 2 indicates that the result of digesting password-and-salt string needs to be run once more through MD5. The first 8 bytes of the final digest are used as an initialization vector for cipher-block chaining mode, while the last 8 bytes are supplied as the key material to the `RC2_CBCPad` algorithm. This algorithm modifies the 64 key bits according to the `effectiveKeyBits` parameter. RSA Security Inc. recommends using values between 40 and 128 bits for the `effectiveKeyBits` parameter. Since only 64 bits of key material are supplied to the algorithm, `effectiveKeyBits` values over 64 bits do not improve security.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a B_RC2_PBE_PARAMS structure:

```
typedef struct {
  unsigned int  effectiveKeyBits;            /* effective key size in bits */
  unsigned char *salt;                     /* pointer to 8-byte salt value */
  unsigned int  iterationCount;                         /* iteration count */
} B_RC2_PBE_PARAMS;
```

This algorithm will accept a maximum of 1024 effective key bits for domestic use and 40 effective key bits for export. RSA Security Inc. recommends a minimum iteration count of 1,000. However, for an additional byte or two of security the iteration should be 2^8 to 2^{16}.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a B_RC2_PBE_PARAMS structure (see above).

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, B_EncryptFinal, B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. You may pass (B_ALGORITHM_OBJ)NULL_PTR for all *randomAlgorithm* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_MD5 and AM_RC2_CBC_ENCRYPT for encryption or AM_RC2_CBC_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_Item that gives the password.

Compatible representation:

AI_MD5WithRC2_CBCPadBER.

Output considerations:

During encryption, this AI pads the output. Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_MD5WithRC2_CBCPadBER

Purpose:

This AI is similar to `AI_MD5WithRC2_CBCPad` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier which includes ASN.1 encoding of the `B_RC2_PBE_PARAMS` structure defined in the description of `AI_MD5WithRC2_CBCPad`. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_MD5WithRC2_CBCPad` or `AI_MD5WithRC2_CBCPadBER`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal is "`42, 134, 72, 134, 247, 13, 1, 5, 6`".

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the MD5 With RC2-CBC password-based encryption algorithm.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than MD5 With RC2-CBC.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_MD5` and `AM_RC2_CBC_ENCRYPT` for encryption or `AM_RC2_CBC_DECRYPT` for

decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item` that gives the password.

Compatible representation:

`AI_MD5WithRC2_CBCPad`.

Output considerations:

During encryption, this AI pads the output. Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_MD5WithRSAEncryption

Purpose:

This AI allows you to perform signature operations that involve the MD5 digest algorithm and the RSA public-key algorithm. The digest of a message is created using the MD5 algorithm and then it is signed using PKCS#1 digital signature algorithm. Other algorithms that can be used for the same purpose are `AI_MD2WithRSAEncryption` and `AI_SHA1WithRSAEncryption`.

Type of information this allows you to use:

the MD5 With RSA signature algorithm that uses the MD5 digest algorithm and the RSA algorithm to create and verify RSA digital signatures as defined in PKCS #1. Note that in order to perform PKCS #1 digital signatures with a 16-byte digest, the RSA key must be at least 360 bits long.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_SignInit`, `B_SignUpdate`, `B_SignFinal`, `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_MD2`, and `AM_RSA_CRT_ENCRYPT`, `AM_RSA_CRT_ENCRYPT_BLIND`, or `AM_RSA_ENCRYPT`, for signature creation; and `AM_RSA_DECRYPT` for signature verification. `AM_RSA_CRT_ENCRYPT_BLIND` performs blinding to protect against timing attacks, whereas `AM_RSA_CRT_ENCRYPT` does not.

Key info types for *keyObject* in B_SignInit:

KI_RSA_CRT, KI_PKCS_RSAPrivate, KI_PKCS_RSAPrivateBER, KI_RSAPrivate or KI_RSAPrivateBSAFE1. Unless you use KI_RSA_CRT for your KI, you must include AM_RSA_ENCRYPT in your application's algorithm chooser.

Key info types for *keyObject* in B_VerifyInit:

KI_RSAPublic, KI_RSAPublicBER, or KI_RSAPublicBSAFE1.

Compatible representation:

AI_MD5WithRSAEncryptionBER.

Output considerations:

The *signature* result of B_SignFinal will be the same size as the RSA key's modulus.

Notes:

Although the RSA signature operation is called "encryption" and the verification operation is called "decryption," the signer uses the digest and the private key and follows the steps needed to decrypt, while the verifier uses the transmitted digest and the public key and follows the steps needed to encrypt.

AI_MD5WithRSAEncryptionBER

Purpose:

This AI is similar to `AI_MD5WithRSAEncryption` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_MD5WithRSAEncryption` or `AI_MD5WithRSAEncryptionBER`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal is "`42, 134, 72, 134, 247, 13, 1, 1, 4`".

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the MD5 With RSA signature algorithm that uses the MD5 digest algorithm and the RSA algorithm to create and verify RSA digital signatures as defined in PKCS #1.

Note that in order to perform PKCS #1 digital signatures with a 16-byte digest, the RSA key must be at least 360 bits long.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than MD5 With RSA Encryption.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_SignInit`, `B_SignUpdate`, `B_SignFinal`, `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_MD2, and AM_RSA_CRT_ENCRYPT, AM_RSA_CRT_ENCRYPT_BLIND, or AM_RSA_ENCRYPT, for signature creation; and AM_RSA_DECRYPT for signature verification. AM_RSA_CRT_ENCRYPT_BLIND performs blinding to protect against timing attacks, whereas AM_RSA_CRT_ENCRYPT does not.

Key info types for *keyObject* in B_SignInit:

KI_RSA_CRT, KI_PKCS_RSAPrivate, KI_PKCS_RSAPrivateBER, KI_RSAPrivate or KI_RSAPrivateBSAFE1. Unless you use KI_RSA_CRT for your KI, you must include AM_RSA_ENCRYPT in your application's algorithm chooser.

Key info types for *keyObject* in B_VerifyInit:

KI_RSAPublic, KI_RSAPublicBER, or KI_RSAPublicBSAFE1.

Compatible representation:

AI_MD5WithRSAEncryption.

Output considerations:

The *signature* result of B_SignFinal will be the same size as the RSA key's modulus.

Notes:

Although the RSA signature operation is called "encryption" and the verification operation is called "decryption," the signer uses the digest and the private key and follows the steps needed to decrypt, while the verifier uses the transmitted digest and the public key and follows the steps needed to encrypt.

AI_MD5WithXOR

Purpose:

This AI is used for encrypting the file keys. This algorithm implements a variant of password-based encryption. The data being encrypted is XORed with a secret key derived from a password, and it can be successfully decrypted only when the correct password is provided. Since the secret key is a 128-bit output of MD5 message digest algorithm, the data being encrypted should be no longer than 128 bits. A description of MD5 can be found in RFC 1321.

Type of information this allows you to use:

the salt and iteration count for the MD5 With "exclusive or" (XOR) password-based encryption algorithm. The salt is concatenated with the password before being digested by MD5, and the iteration count specifies how many times the digest needs to be run. The count of 2 indicates that the result of digesting password-and-salt string needs to be run once more through MD5. The final digest is XORed with the data to obtain the encryption.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_PBE_PARAMS` structure:

```
typedef struct {
  unsigned char *salt;                       /* pointer to 8-byte salt value */
  unsigned int  iterationCount;                             /* iteration count */
} B_PBE_PARAMS;
```

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `B_PBE_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_MD5` and `AM_MD5_RANDOM`.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item` that gives the password.

Compatible representation:

`AI_MD5WithXOR_BER`.

AI_MD5WithXOR_BER

Purpose:

This AI is similar to `AI_MD5WithXOR` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier which includes ASN.1 encoding of the `B_PBE_PARAMS` structure defined in the description of `AI_MD5WithXOR`. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_MD5WithXOR` or `AI_MD5WithXOR_BER`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal is "`42, 134, 72, 134, 247, 13, 1, 5, 9`".

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the MD5 With "exclusive or" (XOR) password-based encryption algorithm, as defined by RSA Security Inc.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than MD5 With XOR.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_MD5` and `AM_MD5_RANDOM`.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item` that gives the password.

Compatible representation:

`AI_MD5WithXOR`.

AI_PKCS_OAEP_RSAPrivate

Purpose:

This AI allows you to decrypt data using the RSA public-key algorithm with the OAEP padding scheme defined in PKCS #1 v2.0. The OAEP padding scheme prevents a theoretical attack on interactive key-establishment protocols that use PKCS #1 v1.5. The parameters of this algorithm include the hash function, mask generator function, and P source function that are explained below. `AI_PKCS_RSAPrivate` provides the PKCS #1 v1.5 version of the RSA private key decryption algorithm. `AI_SET_OAEP_RSAPrivate` provides a different type of OAEP padding scheme defined by the SET specification. See `AI_PKCS_OAEP_RSAPrivateBER` for the same algorithm type with BER encoding.

Type of information this allows you to use:

the RSA algorithm for performing private key decryption with OAEP message padding as defined in PKCS #1 v2.0. When decrypting, this algorithm decodes the data according to the definition of EME-OAEP-Decode as specified in PKCS #1 v2.0.

Format of *info* supplied to B_SetAlgorithmInfo:

either:

`NULL_PTR`. The following parameters are employed when `NULL_PTR` is specified:

PKCS OAEP RSA PARAMETER	DEFAULT VALUE	DEFAULT PARAMS
hashFunc	“sha1”	empty ITEM
maskGenFunc	“mgf1”	empty ITEM
maskGenFuncUnderlyingAlg	“sha1”	empty ITEM
pSourceFunc	“specifiedParameters”	empty ITEM

or:

a pointer to an A_PKCS_OAEP_PARAMS structure:

```
typedef struct {
       unsigned char*     hashFunc;
       ITEM               hashFuncParams;
       unsigned char*     maskGenFunc;
       ITEM               maskGenFuncParams;
       unsigned char*     maskGenFuncUnderlyingAlg;
       ITEM               maskGenFuncUnderlyingAlgParams;
       unsigned char*     pSourceFunc;
       ITEM               pSourceParams;
} A_PKCS_OAEP_PARAMS;
```

The parameters are as follows:

hashFunc determines the digest function. Currently, it may contain a NULL_PTR or a pointer to the null-terminated ASCII string, "sha1". In both cases SHA1 will become the digest function.

maskGenFunc determines the mask generator function. Currently, it may contain a NULL_PTR or a pointer to the null-terminated ASCII string, "mgf1". In both cases MGF1 will become the mask generator function.

maskGenFuncUnderlyingAlg may contain a NULL_PTR or a pointer to the null-terminated ASCII string, "sha1". In both cases SHA1 will become the underlying algorithm.

pSourceFunc is the method for determining the PKCS #1 v2.0 OAEP parameter, P. *pSourceFunc* may contain a NULL_PTR or a pointer to the null-terminated ASCII string, "specifiedParameters". In both cases "specifiedParameters" will become the pSource method.

If *pSourceFunc* is "specifiedParameters" and if *pSourceParams.len* is 0, then P is assumed to be empty. *pSourceParams* may also be initialized to the caller's data as in this example:

```
pSourceParams.len = sizeof(dataObject);
pSourceParams.data = &dataObject;
```

hashFuncParams, *maskGenFuncParams*, and *maskGenFuncUnderlyingAlgParams* are available to provide for future growth. The caller should initialize these parameters as ITEM types as follows:

```
hashFuncParams.len = 0;
hashFuncParams.data = NULL_PTR
maskGenFuncParams.len = 0;
maskGenFuncParams.data = NULL_PTR
maskGenFuncUnderlyingAlgParams.len = 0;
maskGenFuncUnderlyingAlgParams.data = NULL_PTR
```

Failure to properly initialize these parameters may cause errors when they are implemented in future versions of Crypto-C. In this case, the default parameters for *pSourceParams* should be set by the caller as follows:

```
pSourceParams.len = 0;
pSourceParams.data = NULL_PTR;
```

Format of *info* supplied to B_GetAlgorithmInfo:

NULL_PTR.

Crypto-C procedures to use with algorithm object:

B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. You may pass (B_ALGORITHM_OBJ)NULL_PTR for the *randomAlgorithm* argument in B_DecryptUpdate and B_DecryptFinal.

Algorithm methods to include in application's algorithm chooser:

AM_RSA_CRT_DECRYPT or AM_RSA_CRT_DECRYPT_BLIND for decryption. AM_RSA_CRT_DECRYPT_BLIND performs blinding to protect against timing attacks, whereas AM_RSA_CRT_DECRYPT does not. AM_SHA is required for the default *pSource* digest function. It is also required for MGF1 as underlying algorithm.

Key info types for keyObject in B_EncryptInit or B_DecryptInit:

KI_RSA_CRT, KI_PKCS_RSAPrivate, or KI_PKCS_RSAPrivateBER.

Compatible representation:

AI_PKCS_OAEP_RSAPrivateBER.

Output considerations:

The output of decryption will be the same size as the original message.

AI_PKCS_OAEP_RSAPrivateBER

Purpose:

This AI is similar to `AI_PKCS_OAEP_RSAPrivate` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier that includes the hash function, mask generator function, and P source function. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_PKCS_OAEP_RSAPrivate` or `AI_PKCS_OAEP_RSAPrivateBER`. The `OID` for the RSA OAEP encryption, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 1, 7`". The `OID` for the mask function, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 1, 8`". The `OID` for the P source function, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 1, 9`". Also see `AI_PKCS_OAEP_RSAPrivate`.

Type of information this allows you to use:

the RSA algorithm for performing private-key decryption with OAEP message padding as defined in PKCS #1 v2.0. When decrypting, this algorithm decodes the data according to the definition of EME-OAEP-Decode as specified in PKCS #1 v2.0.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than RSAES-OAEP Encryption as specified by PKCS #1 v2.0.

The general ASN.1 syntax for RSAES-OAEP is complicated. The simple DER encoding of the default algorithm is given first, followed by the general syntax.

Simple DER encoding for the default algorithm:

```
  -- Default Algorithm Identifier for RSAES-OAEP.
  -- The DER Encoding of this is in hexadecimal given below.
  -- Notice that the DER encoding of the default parameters
  -- is just an empty sequence.
  -- 30 0D
  --    06 09
  --       2A 86 48 86 F7 0D 01 01 07
  --    30 00
    RSAES-OAEP-Default-Identifier ::=  AlgorithmIdentifier {
     id-RSAES-OAEP,
     {  sha1Identifier,
        mgf1SHA1Identifier,
        pSpecifiedEmptyIdentifier
     }
  }
```

The general syntax is:

```
RSAES-OAEP ::= Sequence {
     algorithm     OBJECT IDENTIFIER (id-RSAES-OAEP),
     parameters    RSAES-OAEP-params
  }
  -- Identifier for PKCS #1 v2.0 OAEP.
  -- The DER for this in hexadecimal is:
  -- 06 09
  --    2A 86 48 86 F7 0D 01 01 07
  --
  id-RSAES-OAEP  OBJECT IDENTIFIER ::= {
     iso(1) member-body(2) us(840) rsadsi(113549)
     pkcs(1) pkcs-1(1) RSAES-OAEP(7)}
  -- Identifier for the PKCS #1 v2.0 mask generation function,
  -- which takes a hash function AlgID as a parameter.
  -- The DER for this in hexadecimal is:
  -- 06 09
  --    2A 86 48 86 F7 0D 01 01 08
  --
  id-mgf1  OBJECT IDENTIFIER ::= {
     iso(1) member-body(2) us(840) rsadsi(113549)
     pkcs(1) pkcs-1(1) mgf1(8)}
```

```
-- The identifier says that the algorithm by which the P
-- string for RSAES-OAEP is generated is by setting it
-- equal to the contents of the OCTET STRING which is
-- the parameter for this AlgorithmIdentifier.
-- The DER for this in hexadecimal is:
-- 06 09
--    2A 86 48 86 F7 0D 01 01 09
--
id-pSpecified  OBJECT IDENTIFIER ::= {
   iso(1) member-body(2) us(840) rsadsi(113549)
   pkcs(1) pkcs-1(1) pSpecified(9)}
-- Identifier for the SHA1 digest function.
-- The DER for this in hexadecimal is:
-- 06 05
--    2B 0E 03 02 1A
--
id-sha1  OBJECT IDENTIFIER ::= {
   iso(1) identified-organization(3) oiw(14) secsig(3)
   algorithms(2) sha1(26) }
-- Syntax of AlgorithmIdentifier.parameters for RSAES-OAEP.
-- Note that the tags in this Sequence are explicit.
-- The DER encoding of DEFAULT values is to omit them.
--
RSAES-OAEP-params ::=  SEQUENCE {
   hashFunc         [0] AlgorithmIdentifier {
        {oaepDigestAlgorithms} }
                    DEFAULT sha1Identifier,
   maskGenFunc      [1] AlgorithmIdentifier {
        {pkcs1MGFAlgorithms} }
                    DEFAULT mgf1SHA1Identifier,
   pSourceFunc      [2] AlgorithmIdentifier {
        {pkcs1PGenAlgorithms} }
                    DEFAULT pSpecifiedEmptyIdentifier
}
-- Algorithm Identifier for SHA1, which is the OAEP default.
--
sha1Identifier ::= AlgorithmIdentifier {
   id-sha1, NULL }
-- Default AlgorithmIdentifier for id-RSAES-OAEP.maskGenFunc.
--
mgf1SHA1Identifier ::= AlgorithmIdentifier {
   id-mgf1, sha1Identifier }
```

```
-- This identifier means that P is an empty string, so the digest
-- of the empty string appears in the RSA block before masking.
--
pSpecifiedEmptyIdentifier ::= AlgorithmIdentifier {
   id-pSpecified, OCTET STRING SIZE (0)
}
```

Format of *info* supplied to B_GetAlgorithmInfo:

pointer to an ITEM structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

The following procedures perform OAEP padding with encryption:

B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. You may pass (B_ALGORITHM_OBJ)NULL_PTR for the *randomAlgorithm* argument in B_DecryptUpdate and B_DecryptFinal.

Algorithm methods to include in application's algorithm chooser:

AM_RSA_CRT_DECRYPT or AM_RSA_CRT_DECRYPT_BLIND for decryption. AM_RSA_CRT_DECRYPT_BLIND performs blinding to protect against timing attacks, whereas AM_RSA_CRT_DECRYPT does not. AM_SHA is required for the default *pSource* digest function. It is also required for MGF1 as underlying algorithm.

Key info types for keyObject in B_EncryptInit or B_DecryptInit:

KI_RSA_CRT, KI_PKCS_RSAPrivate, or KI_PKCS_RSAPrivateBER.

Compatible representation:

AI_PKCS_OAEP_RSAPrivate.

Output considerations:

The output of decryption will be the same size as the original message.

AI_PKCS_OAEP_RSAPublic

Purpose:

This AI allows you to encrypt data using the RSA public-key algorithm with the OAEP padding scheme defined in PKCS #1 v2.0. The OAEP padding scheme prevents a theoretical attack on interactive key-establishment protocols that use PKCS #1 v1.5. The parameters of this algorithm include the hash function, mask generator function and P source function that are explained below. `AI_PKCS_RSAPublic` provides the PKCS #1 v1.5 version of the RSA public-key encryption algorithm. `AI_SET_OAEP_RSAPublic` provides a different type of OAEP padding scheme defined by the SET specification. See `AI_PKCS_OAEP_RSAPublicBER` for the same algorithm type with BER encoding.

Type of information this allows you to use:

the RSA algorithm for performing public-key encryption with OAEP message padding as defined in PKCS #1 v2.0. When encrypting, this algorithm encodes the data according to the definition of EME-OAEP-Encode as specified in PKCS #1 v2.0.

Format of *info* supplied to B_SetAlgorithmInfo:

either:

`NULL_PTR`. The following parameters are employed when `NULL_PTR` is specified:

PKCS OAEP RSA PARAMETER	DEFAULT VALUE	DEFAULT PARAMETERS
hashFunc	“sha1”	empty ITEM
maskGenFunc	“mgf1”	empty ITEM
maskGenFuncUnderlyingAlg	“sha1”	empty ITEM
pSourceFunc	“specifiedParameters”	empty ITEM

or:

a pointer to an `A_PKCS_OAEP_PARAMS` structure:

```
typedef struct {
       unsigned char*     hashFunc;
       ITEM               hashFuncParams;
       unsigned char*     maskGenFunc;
       ITEM               maskGenFuncParams;
       unsigned char*     maskGenFuncUnderlyingAlg;
       ITEM               maskGenFuncUnderlyingAlgParams;
       unsigned char*     pSourceFunc;
       ITEM               pSourceParams;
} A_PKCS_OAEP_PARAMS;
```

The parameters are as follows:

hashFunc determines the digest function. Currently, it may contain a `NULL_PTR` or a pointer to the null-terminated ASCII string, "`sha1`". In both cases SHA1 will become the digest function.

maskGenFunc determines the mask generator function. Currently, it may contain a `NULL_PTR` or a pointer to the null-terminated ASCII string, "`mgf1`". In both cases MGF1 will become the mask generator function.

maskGenFuncUnderlyingAlg may contain a `NULL_PTR` or a pointer to the null-terminated ASCII string, "`sha1`". In both cases SHA1 will become the underlying algorithm.

pSourceFunc is the method for determining the PKCS #1 v2.0 OAEP parameter, P. *pSourceFunc* may contain a `NULL_PTR` or a pointer to the null-terminated ASCII string, "`specifiedParameters`". In both cases "`specifiedParameters`" will become the pSource method.

If *pSourceFunc* is "`specifiedParameters`" and if *pSourceParams.len* is `0`, then P is assumed to be empty. *pSourceParams* may also be initialized to the caller's data as in this example:

```
  pSourceParams.len = sizeof(dataObject);
  pSourceParams.data = &dataObject;
```

hashFuncParams, *maskGenFuncParams*, and *maskGenFuncUnderlyingAlgParams* are available to provide for future growth. The caller should initialize these parameters as `ITEM` types as follows:

```
hashFuncParams.len = 0;
hashFuncParams.data = NULL_PTR
maskGenFuncParams.len = 0;
maskGenFuncParams.data = NULL_PTR
maskGenFuncUnderlyingAlgParams.len = 0;
maskGenFuncUnderlyingAlgParams.data = NULL_PTR
```

Failure to properly initialize these parameters may cause errors when they are implemented in future versions of Crypto-C. In this case, the default parameters for *pSourceParams* should be set by the caller as follows:

```
pSourceParams.len = 0;
pSourceParams.data = NULL_PTR;
```

Format of *info* supplied to B_GetAlgorithmInfo:

NULL_PTR.

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, and B_EncryptFinal.

B_EncryptFinal requires a valid random number generator as a B_ALGORITHM_OBJ in its *randomAlgorithm* argument. PKCS #1 v2.0 does not specify the random number generation method. It is recommended that AI_X962Random_V0 or AI_SHA1Random be initialized with enough seed bytes to produce 160 bits of entropy.

You may pass (B_ALGORITHM_OBJ)NULL_PTR for the *randomAlgorithm* argument in B_EncryptUpdate.

Algorithm methods to include in application's algorithm chooser:

AM_RSA_ENCRYPT.

AM_SHA is required for the default *pSource* digest function and also for the default MGF underlying digest method.

Key info types for keyObject in B_EncryptInit or B_DecryptInit:

KI_RSAPublic or KI_RSAPublicBER.

Compatible representation:

`AI_PKCS_RSAPublicBER`.

Input constraints:

The key's modulus must be at least [(2 * hLen) + 2] bytes longer than the message.

Output considerations:

The output of encryption will be the same size as the key's modulus.

AI_PKCS_OAEP_RSAPublicBER

Purpose:

This AI is similar to `AI_PKCS_OAEP_RSAPublic` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier that includes the hash function, mask generator function, and P source function. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_PKCS_OAEP_RSAPublic` or `AI_PKCS_OAEP_RSAPublicBER`. The `OID` for the RSA OAEP encryption, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 1, 7`". The `OID` for the mask function, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 1, 8`". The `OID` for the P source function, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 1, 9`". Also see `AI_PKCS_OAEP_RSAPublic`.

Type of information this allows you to use:

the RSA algorithm for performing public-key encryption with OAEP message padding as defined in PKCS #1 v2.0. When encrypting, this algorithm encodes the data according to the definition of EME-OAEP-Encode as specified in PKCS #1 v2.0.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than RSAES-OAEP Encryption as specified by PKCS #1 v2.0.

The general ASN.1 syntax for RSAES-OAEP is complicated. Here the simple DER encoding of the default algorithm is given first, followed by the general syntax.

```
-- Default Algorithm Identifier for RSAES-OAEP.
-- The DER Encoding of this is in hexadecimal given below.
-- Notice that the DER encoding of the default parameters
-- is just an empty sequence.
-- 30 0D
--    06 09
--       2A 86 48 86 F7 0D 01 01 07
--    30 00
--
RSAES-OAEP-Default-Identifier ::=  AlgorithmIdentifier {
   id-RSAES-OAEP,
   {  sha1Identifier,
      mgf1SHA1Identifier,
      pSpecifiedEmptyIdentifier
   }
}
```

The general syntax is:

```
RSAES-OAEP ::= Sequence {
   algorithm     OBJECT IDENTIFIER (id-RSAES-OAEP),
   parameters    RSAES-OAEP-params
}
-- Identifier for PKCS #1 v2.0 OAEP.
-- The DER for this in hexadecimal is:
-- 06 09
--    2A 86 48 86 F7 0D 01 01 07
--
id-RSAES-OAEP  OBJECT IDENTIFIER ::= {
   iso(1) member-body(2) us(840) rsadsi(113549)
   pkcs(1) pkcs-1(1) RSAES-OAEP(7)}

-- Identifier for the PKCS #1 v2.0 mask generation function,
-- which takes a hash function AlgID as a parameter.
-- The DER for this in hexadecimal is:
-- 06 09
--    2A 86 48 86 F7 0D 01 01 08
--
```

```
id-mgf1  OBJECT IDENTIFIER ::= {
   iso(1) member-body(2) us(840) rsadsi(113549)
   pkcs(1) pkcs-1(1) mgf1(8)}
-- The identifier says that the algorithm by which the P
-- string for RSAES-OAEP is generated is by setting it
-- equal to the contents of the OCTET STRING which is
-- the parameter for this AlgorithmIdentifier.
-- The DER for this in hexadecimal is:
-- 06 09
--    2A 86 48 86 F7 0D 01 01 09
--
id-pSpecified  OBJECT IDENTIFIER ::= {
   iso(1) member-body(2) us(840) rsadsi(113549)
   pkcs(1) pkcs-1(1) pSpecified(9)}
-- Identifier for the SHA1 digest function.
-- The DER for this in hexadecimal is:
-- 06 05
--    2B 0E 03 02 1A
--
id-sha1  OBJECT IDENTIFIER ::= {
   iso(1) identified-organization(3) oiw(14) secsig(3)
   algorithms(2) sha1(26) }
-- Syntax of AlgorithmIdentifier.parameters for RSAES-OAEP.
-- Note that the tags in this Sequence are explicit.
-- The DER encoding of DEFAULT values is to omit them.
--
RSAES-OAEP-params ::=  SEQUENCE {
   hashFunc         [0] AlgorithmIdentifier {
        {oaepDigestAlgorithms} }
                    DEFAULT sha1Identifier,
   maskGenFunc      [1] AlgorithmIdentifier {
        {pkcs1MGFAlgorithms} }
                    DEFAULT mgf1SHA1Identifier,
   pSourceFunc      [2] AlgorithmIdentifier {
        {pkcs1PGenAlgorithms} }
                    DEFAULT pSpecifiedEmptyIdentifier
}

-- Algorithm Identifier for SHA1, which is the OAEP default.
--
sha1Identifier ::= AlgorithmIdentifier {
   id-sha1, NULL }
-- Default AlgorithmIdentifier for id-RSAES-OAEP.maskGenFunc.
--
```

```
mgf1SHA1Identifier ::= AlgorithmIdentifier {
   id-mgf1, sha1Identifier }
-- This identifier means that P is an empty string, so the digest
-- of the empty string appears in the RSA block before masking.
--
pSpecifiedEmptyIdentifier ::= AlgorithmIdentifier {
   id-pSpecified, OCTET STRING SIZE (0)
}
```

Format of *info* supplied to B_GetAlgorithmInfo:

pointer to an ITEM structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, and B_EncryptFinal.

B_EncryptFinal requires a valid random number generator as a B_ALGORITHM_OBJ in its *randomAlgorithm* argument. PKCS #1 v2.0 does not specify the random number generation method. It is recommended that AI_X962Random_V0 or AI_SHA1Random be initialized with enough seed bytes to produce 160 bits of entropy.

You may pass (B_ALGORITHM_OBJ)NULL_PTR for the *randomAlgorithm* argument in B_EncryptUpdate.

Algorithm methods to include in application's algorithm chooser:

AM_RSA_ENCRYPT.

AM_SHA is required for the default *pSource* digest function and also for the default MGF underlying digest method.

Key info types for keyObject in B_EncryptInit or B_DecryptInit:

KI_RSAPublic and KI_RSAPublicBER may be used to perform RSA encryption or decryption.

Compatible representation:

AI_PKCS_OAEP_RSAPublic.

Input constraints:

The key's modulus must be at least [(2 * hLen) + 2] bytes longer than the message.

Output considerations:

The output of encryption will be the same size as the key's modulus.

AI_PKCS_OAEPRecode

Purpose:

This AI allows you to perform raw or hardware-based encoding or decoding using the PKCS #1 v2.0 OAEP padding scheme. The OAEP padding scheme prevents a theoretical attack on interactive key-establishment protocols that use PKCS #1 v1.5. The parameters of this algorithm include the hash function, mask generator function, and P source function that are explained below. Encrypting with the `AI_PKCS_OAEP_RSAPublic` algorithm is equivalent to first encoding the data with `AI_PKCS_OAEPRecode` using the `B_Encode` routines and then encrypting with `AI_RSAPublic` using the `B_Encrypt` routines. See `AI_PKCS_OAEPRecodeBER` for the same algorithm type with BER encoding.

Type of information this allows you to use:

OAEP message padding as defined in PKCS #1 v2.0. When encoding, this algorithm encodes the data according to the definition of EME-OAEP-Encode as specified in PKCS #1 v2.0. When decoding, this algorithm decodes the data according to the definition of EME-OAEP-Decode. This permits the use of raw or hardware-based RSA encryption with the PKCS #1 v2.0 flavor of Optimal Asymmetric Encryption Padding.

Format of *info* supplied to B_SetAlgorithmInfo:

Either:

`NULL_PTR`.

The following parameters are employed when `NULL_PTR` is specified:

PKCS OAEP RSA PARAMETER	DEFAULT VALUE	DEFAULT PARAMETERS
hashFunc	“sha1”	empty ITEM
maskGenFunc	“mgf1”	empty ITEM
maskGenFuncUnderlyingAlg	“sha1”	empty ITEM
pSourceFunc	“specifiedParameters”	empty ITEM

or:

a pointer to an `A_PKCS_OAEP_PARAMS` structure:

```
typedef struct {
      unsigned char*     hashFunc;
      ITEM               hashFuncParams;
      unsigned char*     maskGenFunc;
      ITEM               maskGenFuncParams;
      unsigned char*     maskGenFuncUnderlyingAlg;
      ITEM               maskGenFuncUnderlyingAlgParams;
      unsigned char*     pSourceFunc;
      ITEM               pSourceParams;
} A_PKCS_OAEP_PARAMS;
```

The parameters are as follows:

hashFunc determines the digest function. Currently, it may contain a `NULL_PTR` or a pointer to the null-terminated ASCII string, “`sha1`”. In both cases SHA1 will become the digest function.

maskGenFunc determines the mask generator function. Currently, it may contain a `NULL_PTR` or a pointer to the null-terminated ASCII string, “`mgf1`”. In both cases MGF1 will become the mask generator function.

maskGenFuncUnderlyingAlg may contain a `NULL_PTR` or a pointer to the null-terminated ASCII string, “`sha1`”. In both cases SHA1 will become the underlying algorithm.

pSourceFunc is the method for determining the PKCS #1 v2.0 OAEP parameter, P. *pSourceFunc* may contain a `NULL_PTR` or a pointer to the null-terminated ASCII string, “`specifiedParameters`”. In both cases “`specifiedParameters`” will become the pSource method.

If *pSourceFunc* is “`specifiedParameters`” and if *pSourceParams.len* is `0`, then P is assumed to be empty. *pSourceParams* may also be initialized to the caller's data as in this example:

```
  pSourceParams.len = sizeof(dataObject);
  pSourceParams.data = &dataObject;
```

hashFuncParams, *maskGenFuncParams*, and *maskGenFuncUnderlyingAlgParams* are available to provide for future growth. The caller should initialize these parameters as `ITEM` types as follows:

```
hashFuncParams.len = 0;
hashFuncParams.data = NULL_PTR
maskGenFuncParams.len = 0;
maskGenFuncParams.data = NULL_PTR
maskGenFuncUnderlyingAlgParams.len = 0;
maskGenFuncUnderlyingAlgParams.data = NULL_PTR
```

Failure to properly initialize these parameters may cause errors when they are implemented in future versions of Crypto-C. In this case, the default parameters for *pSourceParams* should be set by the caller as follows:

```
pSourceParams.len = 0;
pSourceParams.data = NULL_PTR;
```

Format of *info* supplied to B_GetAlgorithmInfo:

NULL_PTR.

Crypto-C procedures to use with algorithm object:

B_EncodeInit, B_EncodeUpdate, B_EncodeFinal, B_DecodeInit, B_DecodeUpdate, and B_DecodeFinal.

The final call to B_EncodeUpdate does not contain message data. Rather, the trailing call to B_EncodeUpdate is included to pass in a number of random seed bytes for the OAEP encoding process. It is recommended that the caller use AI_X962Random_V0 or AI_SHA1Random to generate hLen bytes initialized with 160 bits of entropy. The default digest algorithm for PKCS #1 v2.0 OAEP is SHA1. SHA1 produces a digest of 20 bytes, so hLen for SHA1 is 20 bytes.

B_Decode_Update does not contain an extra call for seed bytes.

Algorithm methods to include in application's algorithm chooser:

AM_SHA is required for the default *pSource* digest function and also for the default MGF underlying digest method.

Compatible representation:

`AI_PKCS_OAEPRecodeBER`.

Input constraints:

The total number of bytes to encode must be at least [(2 * hLen) + 1] bytes long.

Output considerations:

The output of encoding will be an encoded message that is the same size as *`maxPartOutLen`*. (See `B_EncodeUpdate`, `B_EncodeFinal`, `B_DecodeUpdate`, and `B_DecodeFinal`.) The output of the decoding will be the original message.

AI_PKCS_OAEPRecodeBER

Purpose:

This AI is similar to `AI_PKCS_OAEPRecode` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier that includes the hash function, mask generator function, and P source function. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_PKCS_OAEPRecode` or `AI_PKCS_OAEPRecodeBER`. The `OID` for the RSA OAEP encryption, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 1, 7`". The `OID` for the mask function, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 1, 8`". The `OID` for the P source function, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 1, 9`". Also see `AI_PKCS_OAEPRecode`.

Type of information this allows you to use:

OAEP message padding as defined in PKCS #1 v2.0. When encoding, this algorithm encodes the data according to the definition of EME-OAEP-Encode as specified in PKCS #1 v2.0. When decoding, this algorithm decodes the data according to the definition of EME-OAEP-Decode.

This permits the use of raw or hardware-based RSA encryption with the PKCS #1 v2.0 flavor of Optimal Asymmetric Encryption Padding.

Format of info supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than RSAES-OAEP Encryption as specified by PKCS #1 v2.0.

The general ASN.1 syntax for RSAES-OAEP is complicated. The simple DER encoding of the default algorithm is given first, followed by the general syntax.

```
-- Default Algorithm Identifier for RSAES-OAEP.
-- The DER Encoding of this is in hexadecimal given below.
-- Notice that the DER encoding of the default parameters
-- is just an empty sequence.
-- 30 0D
--    06 09
--       2A 86 48 86 F7 0D 01 01 07
--    30 00
--
RSAES-OAEP-Default-Identifier ::=  AlgorithmIdentifier {
   id-RSAES-OAEP,
   {  sha1Identifier,
      mgf1SHA1Identifier,
      pSpecifiedEmptyIdentifier
   }
}
```

The general syntax is:

```
RSAES-OAEP ::= Sequence {
   algorithm     OBJECT IDENTIFIER (id-RSAES-OAEP),
   parameters    RSAES-OAEP-params
}
-- Identifier for PKCS #1 v2.0 OAEP.
-- The DER for this in hexadecimal is:
-- 06 09
--    2A 86 48 86 F7 0D 01 01 07
--
id-RSAES-OAEP  OBJECT IDENTIFIER ::= {
   iso(1) member-body(2) us(840) rsadsi(113549)
   pkcs(1) pkcs-1(1) RSAES-OAEP(7)}
-- Identifier for the PKCS #1 v2.0 mask generation function,
-- which takes a hash function AlgID as a parameter.
-- The DER for this in hexadecimal is:
-- 06 09
--    2A 86 48 86 F7 0D 01 01 08
--
id-mgf1  OBJECT IDENTIFIER ::= {
   iso(1) member-body(2) us(840) rsadsi(113549)
   pkcs(1) pkcs-1(1) mgf1(8)}
```

```
-- The identifier says that the algorithm by which the P
-- string for RSAES-OAEP is generated is by setting it
-- equal to the contents of the OCTET STRING which is
-- the parameter for this AlgorithmIdentifier.
-- The DER for this in hexadecimal is:
-- 06 09
--    2A 86 48 86 F7 0D 01 01 09
--
id-pSpecified  OBJECT IDENTIFIER ::= {
   iso(1) member-body(2) us(840) rsadsi(113549)
   pkcs(1) pkcs-1(1) pSpecified(9)}
-- Identifier for the SHA1 digest function.
-- The DER for this in hexadecimal is:
-- 06 05
--    2B 0E 03 02 1A
--
id-sha1  OBJECT IDENTIFIER ::= {
   iso(1) identified-organization(3) oiw(14) secsig(3)
   algorithms(2) sha1(26) }
-- Syntax of AlgorithmIdentifier.parameters for RSAES-OAEP.
-- Note that the tags in this Sequence are explicit.
-- The DER encoding of DEFAULT values is to omit them.
--
RSAES-OAEP-params ::=  SEQUENCE {
     hashFunc     [0] AlgorithmIdentifier {
          {oaepDigestAlgorithms} }
                  DEFAULT sha1Identifier,
     maskGenFunc  [1] AlgorithmIdentifier {
          {pkcs1MGFAlgorithms} }
                  DEFAULT mgf1SHA1Identifier,
     pSourceFunc  [2] AlgorithmIdentifier {
          {pkcs1PGenAlgorithms} }
                  DEFAULT pSpecifiedEmptyIdentifier
}
-- Algorithm Identifier for SHA1, which is the OAEP default.
--
sha1Identifier ::= AlgorithmIdentifier {
   id-sha1, NULL }
-- Default AlgorithmIdentifier for id-RSAES-OAEP.maskGenFunc.
--
```

```
mgf1SHA1Identifier ::= AlgorithmIdentifier {
   id-mgf1, sha1Identifier }
-- This identifier means that P is an empty string, so the digest
-- of the empty string appears in the RSA block before masking.
--
pSpecifiedEmptyIdentifier ::= AlgorithmIdentifier {
   id-pSpecified, OCTET STRING SIZE (0)
}
```

Format of info supplied to B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_EncodeInit`, `B_EncodeUpdate`, `B_EncodeFinal`, `B_DecodeInit`, `B_DecodeUpdate`, and `B_DecodeFinal`.

The final call to `B_EncodeUpdate` does not contain message data. Rather, the trailing call to `B_EncodeUpdate` is included to pass in a number of random seed bytes for the OAEP encoding process. It is recommended that the caller use `AI_X962Random_V0` or `AI_SHA1Random` to generate hLen bytes initialized with 160 bits of entropy. The default digest algorithm for PKCS #1 v2.0 OAEP is SHA1. SHA1 produces a digest of 20 bytes, so hLen for SHA1 is 20 bytes.

`B_Decode_Update` does not contain an extra call for seed bytes.

Algorithm methods to include in application's algorithm chooser:

`AM_SHA` is required for the default *`pSource`* digest function and also for the default MGF underlying digest method.

Compatible representation:

`AI_PKCS_OAEPRecodeBER`.

Input constraints:

The total number of bytes to encode must be at least [(2 * hLen) + 1] bytes long.

Output considerations:

The output of encoding will be an encoded message that is the same size as *maxPartOutLen*. (See `B_EncodeUpdate`, `B_EncodeFinal`, `B_DecodeUpdate`, and `B_DecodeFinal`.) The output of the decoding will be the original message.

AI_PKCS_RSAPrivate

Purpose:

This AI allows you to decrypt data encrypted using the RSA public-key cryptosystem as defined in PKCS #1.

Type of information this allows you to use:

the RSA algorithm for performing private-key decryption as defined in PKCS #1. When encrypting, this algorithm encodes the data according to block type 01. When decrypting, this algorithm decodes the data from a block type 02.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_RSA_CRT_ENCRYPT` or `AM_RSA_CRT_ENCRYPT_BLIND` for encryption, or `AM_RSA_CRT_DECRYPT` or `AM_RSA_CRT_DECRYPT_BLIND` for decryption.

`AM_RSA_CRT_ENCRYPT_BLIND` and `AM_RSA_CRT_DECRYPT_BLIND` perform blinding to protect against timing attacks, whereas `AM_RSA_CRT_ENCRYPT` and `AM_RSA_CRT_DECRYPT` do not.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_RSA_CRT`, `KI_PKCS_RSAPrivate`, `KI_PKCS_RSAPrivateBER`, or `KI_RSAPrivateBSAFE1`.

Compatible representation:

`AI_PKCS_RSAPrivateBER`, `AI_PKCS_RSAPrivatePEM`.

Input constraints:

The total number of bytes to encrypt may not be more than $k - 11$, where k is the key's modulus size in bytes.

Output considerations:

The output of encryption will be the same size as the key's modulus.

AI_PKCS_RSAPrivateBER

Purpose:

This AI is similar to `AI_PKCS_RSAPrivate` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_PKCS_RSAPrivate`, `AI_PKCS_RSAPrivateBER` or `AI_PKCS_RSAPrivatePEM`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal is “`42, 134, 72, 134, 247, 13, 1, 1, 1`”.

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the RSA algorithm for performing private-key decryption as defined in PKCS #1. When encrypting, this algorithm encodes the data according to block type 01. When decrypting, this algorithm decodes the data from block type 02.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than the RSA algorithm.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_RSA_CRT_ENCRYPT or AM_RSA_CRT_ENCRYPT_BLIND for encryption, or AM_RSA_CRT_DECRYPT or AM_RSA_CRT_DECRYPT_BLIND for decryption.

AM_RSA_CRT_ENCRYPT_BLIND and AM_RSA_CRT_DECRYPT_BLIND perform blinding to protect against timing attacks, whereas AM_RSA_CRT_ENCRYPT and AM_RSA_CRT_DECRYPT do not.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_RSA_CRT, KI_PKCS_RSAPrivate, KI_PKCS_RSAPrivateBER, or KI_RSAPrivateBSAFE1.

Compatible representation:

AI_PKCS_RSAPrivate, AI_PKCS_RSAPrivatePEM.

Input constraints:

The total number of bytes to encrypt may not be more than $k - 11$, where k is the key's modulus size in bytes.

Output considerations:

The output of encryption will be the same size as the key's modulus.

AI_PKCS_RSAPrivatePEM

Purpose:

This AI is similar to `AI_PKCS_RSAPrivate` except that it uses the Privacy Enhanced Mail (PEM) format. This AI allows you to parse and create PEM algorithm identifiers such as those used in the PEM protocol. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_PKCS_RSAPrivate`, `AI_PKCS_RSAPrivateBER` or `AI_PKCS_RSAPrivatePEM`.

Type of information this allows you to use:

an RFC 1423 identifier that specifies the RSA algorithm for performing private key decryption as defined in PKCS #1. When encrypting, this algorithm encodes the data according to block type 01. When decrypting, this algorithm decodes the data from block type 02.

This algorithm info type is intended to process the asymmetric encryption identifier in a MIC-Info and Key-Info field in a PEM-encapsulated header.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a null-terminated string (`char *`) that gives the RSA identifier, for example, “`RSA`”. Space and tab characters are removed from the string before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier does not specify the RSA algorithm.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a null-terminated string that gives the RSA identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_RSA_CRT_ENCRYPT or AM_RSA_CRT_ENCRYPT_BLIND for encryption, or AM_RSA_CRT_DECRYPT or AM_RSA_CRT_DECRYPT_BLIND for decryption. AM_RSA_CRT_ENCRYPT_BLIND and AM_RSA_CRT_DECRYPT_BLIND perform blinding to protect against timing attacks, whereas AM_RSA_CRT_ENCRYPT and AM_RSA_CRT_DECRYPT do not.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_RSA_CRT, KI_PKCS_RSAPrivate, KI_PKCS_RSAPrivateBER or KI_RSAPrivateBSAFE1.

Compatible representation:

AI_PKCS_RSAPrivate, AI_PKCS_RSAPrivateBER.

Input constraints:

The total number of bytes to encrypt may not be more than *k – 11*, where *k* is the keys modulus size in bytes.

Output considerations:

The output of encryption will be the same size as the key's modulus.

AI_PKCS_RSAPublic

Purpose:

This AI allows you to encrypt data using the RSA public-key cryptosystem as defined in PKCS #1.

Type of information this allows you to use:

the RSA algorithm for performing public-key encryption as defined in PKCS #1. When encrypting, this algorithm encodes the data according to block type 02. When decrypting, this algorithm decodes the data from block type 01.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, and `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. Note that `B_EncryptUpdate` and `B_EncryptFinal` require a random algorithm. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for the *`randomAlgorithm`* argument in `B_DecryptUpdate` and `B_DecryptFinal`.

Algorithm methods to include in application's algorithm chooser:

`AM_RSA_ENCRYPT` for encryption or `AM_RSA_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_RSAPublic`, `KI_RSAPublicBER`, or `KI_RSAPublicBSAFE1`.

Compatible representation:

`AI_PKCS_RSAPublicBER`, `AI_PKCS_RSAPublicPEM`.

Input constraints:

The total number of bytes to encrypt may not be more than $k - 11$, where k is the key's modulus size in bytes.

Output considerations:

The output of encryption will be the same size as the key's modulus.

AI_PKCS_RSAPublicBER

Purpose:

This AI is similar to `AI_PKCS_RSAPublic` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_PKCS_RSAPublic`, `AI_PKCS_RSAPublicBER` or `AI_PKCS_RSAPublicPEM`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal is "`42, 134, 72, 134, 247, 13, 1, 1, 1`".

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the RSA algorithm for performing public-key encryption as defined in PKCS #1. When encrypting, this algorithm encodes the data according to block type 02. When decrypting, this algorithm decodes the data from block type 01.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than the RSA algorithm.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. Note that `B_EncryptUpdate` and `B_EncryptFinal` require a random algorithm. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for the *`randomAlgorithm`* argument in `B_DecryptUpdate` and `B_DecryptFinal`.

Algorithm methods to include in application's algorithm chooser:

`AM_RSA_ENCRYPT` for encryption and `AM_RSA_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_RSAPublic`, `KI_RSAPublicBER` or `KI_RSAPublicBSAFE1`.

Compatible representation:

`AI_PKCS_RSAPublic`, `AI_PKCS_RSAPublicPEM`.

Input constraints:

The total number of bytes to encrypt may not be more than $k - 11$, where k is the key's modulus size in bytes.

Output considerations:

The output of encryption will be the same size as the key's modulus.

AI_PKCS_RSAPublicPEM

Purpose:

This AI is similar to `AI_PKCS_RSAPublic` except that it uses the Privacy Enhanced Mail (PEM) format. This AI allows you to parse and create PEM algorithm identifiers such as used in PEM protocol. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_PKCS_RSAPublic`, `AI_PKCS_RSAPublicBER`, or `AI_PKCS_RSAPublicPEM`.

Type of information this allows you to use:

an RFC 1423 identifier that specifies the RSA algorithm for performing public key encryption as defined in PKCS #1. When encrypting, this algorithm encodes the data according to block type 02. When decrypting, this algorithm decodes the data from block type 01. This algorithm info type is intended to process the asymmetric encryption identifier in a MIC-Info and Key-Info field in a PEM encapsulated header.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a null-terminated string (`char *`) that gives the RSA identifier. For example, "`RSA`". Space and tab characters are removed from the string before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier does not specify the RSA algorithm.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a null-terminated string that gives the RSA identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. Note that `B_EncryptUpdate` and `B_EncryptFinal` require a random algorithm. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for the *`randomAlgorithm`* argument in `B_DecryptUpdate` and `B_DecryptFinal`.

Algorithm methods to include in application's algorithm chooser:

AM_RSA_ENCRYPT for encryption or AM_RSA_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_RSAPublic, KI_RSAPublicBER, or KI_RSAPublicBSAFE1.

Compatible representation:

AI_PKCS_RSAPublic, AI_PKCS_RSAPublicBER.

Input constraints:

The total number of bytes to encrypt may not be more than *k – 11*, where *k* is the key's modulus size in bytes.

Output considerations:

The output of encryption will be the same size as the key's modulus.

AI_RC2_CBC

Purpose:

This AI allows you to perform RC2 encryption or decryption in CBC mode with an 8-byte initialization vector. During encryption, this algorithm does not pad the output. Thus, you must provide input that is a multiple of 8 bytes. See AI_RC2_CBCPad for the same algorithm with padding. RC2 is a variable-key-size block cipher, which means that the key can be anywhere between 1 and 128 bytes. The larger the key, the greater the security. RC2 is described in RFC 2268, and the CBC mode is similar to the one described in RFC 2040.

Other algorithms that can be used for encryption/decryption in CBC mode without padding are AI_DES_CBC_IV8, AI_DES_EDE3_CBC_IV8, AI_DESX_CBC_IV8, and AI_RC5_CBC.

Type of information this allows you to use:

an effective key size and an 8-byte initialization vector for the RC2-CBC encryption algorithm.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an A_RC2_CBC_PARAMS structure:

```
typedef struct {
  unsigned int   effectiveKeyBits;          /* effective key size in bits */
  unsigned char *iv;                           /* initialization vector */
} A_RC2_CBC_PARAMS;
```

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an A_RC2_CBC_PARAMS structure (see above).

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, B_EncryptFinal, B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. You may pass (B_ALGORITHM_OBJ)NULL_PTR for all *randomAlgorithm* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_RC2_CBC_ENCRYPT` for encryption and `AM_RC2_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_8Byte`, `KI_Item`, `KI_RC2WithBSAFE1Params`, or `KI_RC2_BSAFE1`.

Input constraints:

During encryption, this algorithm does not pad the output. Thus, you must provide input that is a multiple of 8 bytes.

Token-based algorithm methods:

`AI_RC2_CBC` may be used to access the hardware-related algorithm methods `AM_TOKEN_RC2_CBC_ENCRYPT` and `AM_TOKEN_RC2_CBC_DECRYPT`, for use with BHAPI.

Token-based key info types:

When used with one of the hardware algorithm methods listed above, `AI_RC2_CBC` should be used with `KI_Token` or `KI_ExtendedToken`.

AI_RC2_CBC_BSAFE1

Purpose:

Deprecated. This AI is included only for backward compatibility.

Type of information this allows you to use:

the encryption type parameter (pad, pad with checksum, or raw) for the RC2 encryption algorithm as defined by BSAFE 1.x. Note that the RC2 algorithm is the same as SX1.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_BSAFE1_ENCRYPTION_PARAMS` structure:

```
typedef struct {
  int encryptionType;                                /* encryption type */
} B_BSAFE1_ENCRYPTION_PARAMS;
```

`encryptionType` should be set to `B_BSAFE1_PAD` for pad mode, `B_BSAFE1_PAD_CHECKSUM` for pad with checksum mode, or `B_BSAFE1_RAW` for raw mode.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `B_BSAFE1_ENCRYPTION_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application’s algorithm chooser:

`AM_RC2_CBC_ENCRYPT` for encryption and `AM_RC2_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_RC2WithBSAFE1Params` or `KI_RC2_BSAFE1`.

Input constraints:

During encryption, this algorithm does not pad the output. Thus, you must provide input that is a multiple of 8 bytes.

Output considerations:

In pad mode, the total number of output bytes from encryption can be as many as 8 more than the total input. In pad with checksum mode, the total number of output bytes from encryption can be as many as 16 bytes more than the total input.

AI_RC2_CBCPad

Purpose:

This AI allows you to perform RC2 encryption or decryption in CBC mode with an 8-byte initialization vector. This algorithms pads as described in PKCS #5, so the input data does not have to be a multiple of 8 bytes. RC2 is a variable-key-size block cipher which means that the key can be anywhere between 1 and 128 bytes. The larger the key, the greater the security. RC2 is described in RFC 2268, and CBC mode is similar to the one described in RFC 2040.

Other algorithms that can be used for encryption/decryption in CBC mode with padding are `AI_DES_CBCPadIV8`, `AI_DES_EDE3_CBCPadIV8`, `AI_DESX_CBCPadIV8`, and `AI_RC5_CBCPad`.

Type of information this allows you to use:

an 8-byte initialization vector and effective key size for the RC2-CBC With Padding encryption algorithm.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `A_RC2_CBC_PARAMS` structure:

```
typedef struct {
  unsigned int   effectiveKeyBits;           /* effective key size in bits */
  unsigned char *iv;                               * initialization vector */
} A_RC2_CBC_PARAMS;
```

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `A_RC2_CBC_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application’s algorithm chooser:

AM_RC2_CBC_ENCRYPT for encryption and AM_RC2_CBC_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_8Byte, KI_Item, KI_RC2WithBSAFE1Params, or KI_RC2_BSAFE1.

Compatible representation:

AI_RC2_CBCPadBER, AI_RC2_CBCPadPEM.

Output considerations:

During encryption, this AI pads the output. Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_RC2_CBCPadBER

Purpose:

This AI is similar to AI_RC2_CBCPad except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call B_SetAlgorithmInfo to initialize an algorithm object from the encoded algorithm identifier which includes ASN.1 encoding of the A_RC2_CBC_PARAMS structure defined in the description of AI_RC2_CBCPad. You call B_GetAlgorithmInfo with this AI to create an encoded algorithm identifier from an algorithm object that was created using AI_RC2_CBCPad, AI_RC2_CBCPadBER or AI_RC2_CBCPadPEM. The OID for this algorithm, excluding the tag and length bytes, in decimal is "42, 134, 72, 134, 247, 13, 3, 2".

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the RC2-CBC With Padding encryption algorithm.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an ITEM structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. B_SetAlgorithmInfo returns BE_WRONG_ALGORITHM_INFO if the algorithm identifier specifies an algorithm other than RC2-CBC With Padding.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an ITEM structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, B_EncryptFinal, B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. You may pass (B_ALGORITHM_OBJ)NULL_PTR for all *randomAlgorithm* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_RC2_CBC_ENCRYPT for encryption and AM_RC2_CBC_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_8Byte`, `KI_Item`, `KI_RC2WithBSAFE1Params`, or `KI_RC2_BSAFE1`.

Compatible representation:

`AI_RC2_CBCPad`, `AI_RC2_CBCPadPEM`.

Output considerations:

During encryption, this AI pads the output. Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_RC2_CBCPadPEM

Purpose:

This AI is similar to `AI_RC2_CBCPad` except that it uses the Privacy Enhanced Mail (PEM) format. This AI allows you to parse and create PEM algorithm identifiers such as used in PEM protocol. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier, which includes PEM encoding of the `A_RC2_CBC_PARAMS` structure defined in the description of `AI_RC2_CBCPad`. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_RC2_CBCPad`, `AI_RC2_CBCPadBER`, or `AI_RC2_CBCPadPEM`.

Type of information this allows you to use:

an RFC 1423-style identifier that specifies the RC2-CBC With Padding encryption algorithm. This algorithm info type is intended to process the value of a DEK-Info field in a PEM-encapsulated header.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a null-terminated string (char *) that gives the RC2-CBC identifier and parameters. For example, "`RC2-CBC,BAgRIjNEVWZ3iA==`". Space and tab characters are removed from the string before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an identifier other than RC2-CBC.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a null-terminated string that gives the RC2-CBC identifier and parameters.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_RC2_CBC_ENCRYPT` for encryption and `AM_RC2_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_8Byte`, `KI_Item`, `KI_RC2WithBSAFE1Params`, or `KI_RC2_BSAFE1`.

Compatible representation:

`AI_RC2_CBCPad`, `AI_RC2_CBCPadBER`.

Output considerations:

During encryption, this AI pads the output. Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_RC4

Purpose:

This AI allows you to perform RC4 encryption and decryption. The description of the RC4 stream cipher can be found in B. Schneier's book *Applied Cryptography*. Because it is a stream cipher, there are no restrictions on the length of input data. A similar algorithm that allows you to authenticate the encrypted data is `AI_RC4WithMAC`.

Type of information this allows you to use:

the RC4 encryption algorithm.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Due to the nature of the RC4 algorithm, security is compromised if multiple data blocks are encrypted with the same RC4 key. Therefore, `B_EncryptUpdate` cannot be called after `B_EncryptFinal`. To begin an encryption operation for a new data block, you must call `B_EncryptInit` and supply a new key.

Algorithm methods to include in application's algorithm chooser:

`AM_RC4_ENCRYPT` for encryption and `AM_RC4_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item` that gives the address and length of the RC4 key.

Compatible representation:

AI_RC4_BER.

Token-based algorithm methods:

AI_RC4 may be used to access the hardware-related algorithm methods AM_TOKEN_RC4_ENCRYPT and AM_TOKEN_RC4_DECRYPT, for use with BHAPI.

Token-based key info types:

When used with one of the hardware algorithm methods listed above, AI_RC4 should be used with KI_Token or KI_ExtendedToken.

AI_RC4_BER

Purpose:

This AI is similar to `AI_RC4` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_RC4` or `AI_RC4_BER`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal is "`42, 134, 72, 134, 247, 13, 3, 4`".

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the RC4 encryption algorithm.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than RC4.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Due to the nature of the RC4 algorithm, security is compromised if multiple data blocks are encrypted with the same RC4 key. Therefore, `B_EncryptUpdate` cannot be called after `B_EncryptFinal`. To begin an encryption operation for a new data block, you must call `B_EncryptInit` and supply a new key.

Algorithm methods to include in application's algorithm chooser:

`AM_RC4_ENCRYPT` for encryption and `AM_RC4_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item` that gives the address and length of the RC4 key.

Compatible representation:

`AI_RC4`.

AI_RC4WithMAC

Purpose:

This AI implements a stream cipher with a simple tamper-detection message authentication code based on `AI_MAC`. When applied to a plaintext buffer of `n` bytes, it produces a ciphertext of `n` bytes using the same algorithm as `AI_RC4`, and then it appends a MAC of `macLen` bytes. You can find a description of `AI_RC4` in B. Schneier's *Applied Cryptography*. For more information about `AI_MAC`, see page 96.

Type of information this allows you to use:

the RC4 With MAC encryption algorithm. The MAC is computed using `AI_MAC` by first passing the key to `AI_MAC`, then the plaintext, and finally a block of macLen zero bytes. The resulting value from `AI_MAC` is appended to the ciphertext. For decryption, the MAC value is checked.

The key passed to both `AI_RC4` and `AI_MAC` is created by appending the salt bytes to the end of the key passed to `B_EncryptInit` or `B_DecryptInit`. That is, for this AI, the RC4 key depends on the salt as well as the key object passed to the `Init` routine.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_RC4_WITH_MAC_PARAMS` structure:

```
typedef struct {
  ITEM         salt;                               /* variable-length salt */
  unsigned int macLen;                         /* length to use for MAC value */
} B_RC4_WITH_MAC_PARAMS;
```

The *`salt`* `ITEM` supplies the salt value that is appended to the key, where the `ITEM`'s *`data`* points to an unsigned byte array and the `ITEM`'s *`len`* gives its length. If the length is zero, no salt is appended to the key, and the `ITEM`'s *`data`* is ignored. *`macLen`* has a minimum of 2 and maximum of 16.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `B_RC4_WITH_MAC_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, B_EncryptFinal, B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. B_DecryptFinal returns BE_INPUT_DATA if the MAC does not match. You may pass (B_ALGORITHM_OBJ)NULL_PTR for all *randomAlgorithm* arguments.

Due to the nature of the RC4 algorithm, security is compromised if multiple data blocks are encrypted with the same RC4 key. Therefore, B_EncryptUpdate cannot be called after B_EncryptFinal. To begin an encryption operation for a new data block, you must call B_EncryptInit and supply a new key.

Algorithm methods to include in application's algorithm chooser:

AM_RC4_WITH_MAC_ENCRYPT for encryption, or AM_RC4_WITH_MAC_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_Item that gives the address and length of the RC4 key.

Compatible representation:

AI_RC4WithMAC_BER.

Output considerations:

The total number of output bytes from encryption will be *macLen* bytes more than the input.

Token-based algorithm methods:

AI_RC4WithMAC may be used to access the hardware-related algorithm methods AM_TOKEN_RC4_ENCRYPT and AM_TOKEN_RC4_DECRYPT, for use with BHAPI.

Token-based key info types:

When used with one of the hardware algorithm methods listed above, AI_RC4WithMac should be used with KI_Token or KI_ExtendedToken.

AI_RC4WithMAC_BER

Purpose:

This AI is similar to `AI_RC4WithMAC` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier which includes ASN.1 encoding of the `B_RC4_WITH_MAC_PARAMS` structure defined in the description of `AI_RC4WithMAC`. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_RC4WithMAC` or `AI_RC4WithMAC_BER.` The `OID` for this algorithm, excluding the tag and length bytes, in decimal is “`42, 134, 72, 134, 247, 13, 3, 5`”.

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the RC4 With MAC encryption algorithm.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than RC4.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. `B_DecryptFinal` returns `BE_INPUT_DATA` if the MAC does not match. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Due to the nature of the RC4 algorithm, security is compromised if multiple data blocks are encrypted with the same RC4 key. Therefore, `B_EncryptUpdate` cannot be called after `B_EncryptFinal`. To begin an encryption operation for a new data block, you must call `B_EncryptInit` and supply a new key.

Algorithm methods to include in application's algorithm chooser:

`AM_RC4_WITH_MAC_ENCRYPT` for encryption or `AM_RC4_WITH_MAC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item` that gives the address and length of the RC4 key.

Compatible representation:

`AI_RC4WithMAC`.

Output considerations:

The total number of output bytes from encryption will be *`macLen`* bytes more than the input.

AI_RC5_CBC

Purpose:

This AI allows you to perform RC5 encryption and decryption in CBC mode with an 8-byte initialization vector and a variable number of rounds, as defined in RFC 2040. Since `AI_RC5_CBC` does not pad, the total number of input bytes must be a multiple of 8 bytes. See `AI_RC5_CBCPad` for the same algorithm with padding.

Other algorithms that can be used for encryption/decryption in CBC mode without padding are `AI_DES_CBC_IV8`, `AI_DES_EDE3_CBC_IV8`, `AI_DESX_CBC_IV8`, and `AI_RC2_CBC`.

Type of information this allows you to use:

a version number, a rounds count, a word size, and an 8-byte initialization vector for the RC5 32/r/b CBC encryption algorithm.

Note: To implement the RC5 cipher with a word size other than 32 bits, you should use `AI_FeedbackCipher`.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `A_RC5_CBC_PARAMS` structure:

```
typedef struct {
  unsigned int   version;                    /* currently 1.0 defined 0x10 */
  unsigned int   rounds;                     /* number of rounds (0 - 255) */
  unsigned int   wordSizeInBits;                 /* AI_RC5_CBC requires 32 */
  unsigned char *iv;                    /* initialization vector (8 bytes) */
} A_RC5_CBC_PARAMS;
```

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `A_RC5_CBC_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_RC5_CBC_ENCRYPT` for encryption and `AM_RC5_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item`, that gives the address and length of the RC5 key.

Input Constraints:

During encryption, this algorithm does not pad the output. Thus, you must provide input that is a multiple of 8 bytes.

Token-based algorithm methods:

`AI_RC5_CBC` may be used to access the hardware-related (BHAPI) algorithm methods `AM_TOKEN_RC5_CBC_ENCRYPT` and `AM_TOKEN_RC5_CBC_DECRYPT`.

Token-based key info types:

When used with one of the hardware algorithm methods listed above, `AI_RC5_CBC` should be used with `KI_Token` or `KI_ExtendedToken`.

AI_RC5_CBCPad

Purpose:

This AI allows you to perform RC5 encryption or decryption in CBC mode with an 8-byte initialization vector and a variable number of rounds, as defined in RFC 2040. This algorithms pads, so the input data does not have to be a multiple of 8 bytes.

Other algorithms that can be used for encryption/decryption in CBC mode with padding are `AI_DES_CBCPadIV8`, `AI_DES_EDE3_CBCPadIV8`, `AI_DESX_CBCPadIV8`, and `AI_RC2_CBCPad`.

Type of information this allows you to use:

a version number, a rounds count, a word size, and an 8-byte initialization vector for the RC5 32/r/b CBC encryption algorithm.

Note: To implement the RC5 cipher with a word size other than 32 bits, use `AI_FeedbackCipher`.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `A_RC5_CBC_PARAMS` structure:

```
typedef struct {
  unsigned int   version;                      /* currently 1.0 defined 0x10 */
  unsigned int   rounds;                       /* number of rounds (0 - 255) */
  unsigned int   wordSizeInBits;                /* AI_RC5_CBCPad requires 32 */
  unsigned char *iv;                     /* initialization vector (8 bytes) */
} A_RC5_CBC_PARAMS;
```

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `A_RC5_CBC_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *randomAlgorithm* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_RC5_CBC_ENCRYPT` for encryption and `AM_RC5_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item`, that gives the address and length of the RC5 key.

Compatible representation:

`AI_RC5_CBCPadBER`.

Output considerations:

During encryption, this AI pads the output. Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_RC5_CBCPadBER

Purpose:

This AI is similar to AI_RC5_CBCPad except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call B_SetAlgorithmInfo to initialize an algorithm object from the encoded algorithm identifier, which includes ASN.1 encoding of the A_RC5_CBC_PARAMS structure defined in the description of AI_RC5_CBCPad. You call B_GetAlgorithmInfo with this AI to create an encoded algorithm identifier from an algorithm object that was created using AI_RC5_CBCPad, AI_RC5_CBCPadBER or AI_RC5_CBCPadPEM. The OID for this algorithm, excluding the tag and length bytes, in decimal is "42, 134, 72, 134, 247, 13, 3, 9".

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the RC5 32/r/b CBC encryption algorithm.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an ITEM structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. B_SetAlgorithmInfo returns BE_WRONG_ALGORITHM_INFO if the algorithm identifier specifies an algorithm other than RC5-CBC With Padding.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an ITEM structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, B_EncryptFinal, B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. You may pass (B_ALGORITHM_OBJ)NULL_PTR for all *randomAlgorithm* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_RC5_CBC_ENCRYPT for encryption and AM_RC5_CBC_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item`, which gives the address and length of the RC5 key.

Compatible representation:

`AI_RC5_CBCPad`.

Output considerations:

During encryption, this AI pads the output. Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_RC6_CBC

Purpose:

This AI allows you to perform RC6 encryption and decryption in CBC mode with a 16-byte initialization vector and 20 rounds, as defined in the AES submission. Since AI_RC6_CBC does not pad, the total number of input bytes must be a multiple of 16 bytes. See AI_RC6_CBCPad for the same algorithm with padding.

Type of information this allows you to use:

a rounds count (currently only 20 is a valid round count), and a 16-byte initialization vector for the RC6 CBC encryption algorithm.

Format of info supplied to B_SetAlgorithmInfo:

pointer to an A_RC6_CBC_PARAMS structure:

```
typedef struct {
  unsigned int rounds;                         /* number of rounds (20) */
  unsigned char *iv;                /* initialization vector (16 bytes) */
} A_RC6_CBC_PARAMS;
```

Note: Although one of the parameters is the number of rounds, Crypto-C currently accepts only 20. Future versions may accept other round counts.

Format of info returned by B_GetAlgorithmInfo:

pointer to an A_RC6_CBC_PARAMS structure (see above).

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, B_EncryptFinal, B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. You may pass (B_ALGORITHM_OBJ)NULL_PTR for all *randomAlgorithm* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_RC6_CBC_ENCRYPT for encryption and AM_RC6_CBC_DECRYPT for decryption.

Key info types for keyObject in B_EncryptInit or B_DecryptInit:

`KI_Item`, that gives the address and length of the RC6 key.

Input Constraints:

During encryption, this algorithm does not pad the output. Thus, you must provide input that is a mulitple of 16 bytes.

AI_RC6_CBCPad

Purpose:

This AI allows you to perform RC6 encryption and decryption in CBC mode with a 16-byte initialization vector and 20 rounds, as defined in the AES submission. This algorithm pads, so the total number of input bytes does not have to be a multiple of 16 bytes.

Type of information this allows you to use:

a rounds count (currently only 20 is a valid round count), and a 16-byte initialization vector for the RC6 CBC encryption algorithm.

Format of info supplied to B_SetAlgorithmInfo:

pointer to an `A_RC6_CBC_PARAMS` structure:

```
typedef struct {
  unsigned int rounds;     /* number of rounds (20) */
  unsigned char *iv; /* initialization vector (16 bytes) */
} A_RC6_CBC_PARAMS;
```

Note: Although one of the parameters is number of rounds, Crypto-C currently accepts only 20. Future versions may accept other round counts.

Format of info returned by B_GetAlgorithmInfo:

pointer to an `A_RC6_CBC_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_RC6_CBC_ENCRYPT` for encryption and `AM_RC6_CBC_DECRYPT` for decryption.

Key info types for keyObject in B_EncryptInit or B_DecryptInit:

`KI_Item`, that gives the address and length of the RC6 key.

Output Considerations:

During encryption, this algorithm pads the output. Thus, the total number of output bytes from encryption can be as many as 16 bytes more than the total input.

AI_RESET_IV

Purpose:

This AI allows you to change the initialization vector (IV) for a cipher object created with `AI_FeedbackCipher` without the need to create a new algorithm object or bind in a new key. This increases the performance of applications that have a long-lived symmetric key (for example, a DES key) that is used to encrypt many blocks or messages, each of which has a unique IV.

A similar AI that can be used to change the IV of a CBC mode cipher is `AI_CBC_IV8`.

Type of Information this allows you to use:

a new initialization vector to reset an encryption/decryption object defined with `AI_FeedbackCipher`.

Format of *info* supplied to B_SetAlgorithmInfo

a pointer to an `ITEM` of length equal to the old initialization vector.

Format of *info* returned by B_GetAlgorithmInfo returns

a pointer to an `ITEM`.

Algorithm methods to include in chooser

none.

Note: Reinitialization may be done anytime after an encryption/decryption session has been set with its algorithm info. It takes effect after the next initialization. It may be used in conjunction with a new key to start a new encryption/decryption session.

AI_RFC1113Recode

Purpose:

This AI allows you to convert data from binary format to ASCII characters, and vice versa, as defined by RFC 1113.

Type of information this allows you to use:

the binary to printable recoding algorithm as defined by RFC 1113; see also RFC 1421 for updated version of the standard.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_EncodeInit`, `B_EncodeUpdate`, `B_EncodeFinal`, `B_DecodeInit`, `B_DecodeUpdate`, and `B_DecodeFinal`.

Output considerations:

When encoding, for each 3 bytes of input there are 4 bytes of output. When decoding, for each 4 bytes of input there are 3 bytes of output.

AI_RSAKeyGen

Purpose:

This AI allows you to specify the parameters for generating an RSA public/private key pair as defined in PKCS #1. You may also use AI_RSAStrongKeyGen to generate RSA key pairs.

Type of information this allows you to use:

the parameters for generating an RSA public/private key pair as defined in PKCS #1, where the modulus size and public exponent are specified.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an A_RSA_KEY_GEN_PARAMS structure:

```
typedef struct {
  unsigned int modulusBits;                    /* size of modulus in bits */
  ITEM         publicExponent;                   /* fixed public exponent */
} A_RSA_KEY_GEN_PARAMS;
```

The *publicExponent* ITEM supplies an integer in canonical format, where the ITEM's *data* points to an unsigned byte array—most significant byte first—and the ITEM's *len* gives its length. All leading zeros are stripped from the integer before it is copied to the algorithm object.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an A_RSA_KEY_GEN_PARAMS structure (see above). All leading zeros have been stripped from the *publicExponent* integer.

Crypto-C procedures to use with algorithm object:

B_GenerateInit and B_GenerateKeypair. B_GenerateKeypair sets the *publicKey* key object with the KI_RSAPublic information and the *privateKey* key object with the KI_PKCS_RSAPrivate information. You must pass an initialized random algorithm to B_GenerateKeypair.

Algorithm methods to include in application’s algorithm chooser:

`AM_RSA_KEY_GEN`.

AI_RSAPrivate

Purpose:

This AI allows you to encrypt data using the raw RSA algorithm. You can find the description of this algorithm in B. Schneier's *Applied Cryptography*.

`AI_RSAPrivate` is different from `AI_PKCS_RSAPrivate` because the former allows you to encrypt raw data, while the latter encrypts data in PKCS #1 format.

Because this algorithm does not pad, the total number of input bytes must be a multiple of the key's modulus size in bytes. Your application is responsible for padding the data as appropriate. Also, each modulus-size block of input, interpreted as an integer with the most significant byte first, must be numerically less than the key's modulus. To do this, divide your data into blocks that are one byte smaller than the modulus, and prepend one byte of zeros to each block.

To perform RSA encryption you can also use `AI_PKCS_RSAPrivate` and `AI_SET_OAEP_RSAPrivate`. But you can use `AI_RSAPrivate` only if the data will be decrypted with `AI_RSAPublic`.

Type of information this allows you to use:

the RSA algorithm for performing raw private-key encryption.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_RSA_CRT_ENCRYPT or AM_RSA_CRT_ENCRYPT_BLIND for encryption, or AM_RSA_CRT_DECRYPT or AM_RSA_CRT_DECRYPT_BLIND for decryption.

AM_RSA_CRT_ENCRYPT_BLIND and AM_RSA_CRT_DECRYPT_BLIND perform blinding to protect against timing attacks, whereas AM_RSA_CRT_ENCRYPT and AM_RSA_CRT_DECRYPT do not.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_RSA_CRT, KI_PKCS_RSAPrivate, KI_PKCS_RSAPrivateBER, or KI_RSAPrivateBSAFE1.

Input constraints:

Because this algorithm does not pad, the total number of input bytes must be a multiple of the key's modulus size in bytes. Also, each modulus-size block of input, interpreted as an integer with the most significant byte first, must be numerically less than the key's modulus.

Token-based algorithm methods:

AI_RSAPrivate may include the hardware algorithm method AM_TOKEN_RSA_CRT_ENCRYPT or AM_TOKEN_RSA_CRT_DECRYPT in the algorithm chooser, for use with BHAPI.

Token-based key info types:

When used with one of the hardware algorithm methods described, AI_RSAPrivate should be used with KI_Token or KI_KeypairToken.

AI_RSAPrivateBSAFE1

Purpose:

Deprecated. This AI is included only for backward compatibility.

Type of information this allows you to use:

the encryption type parameter (pad, pad with checksum, or raw) for performing RSA private-key encryption as defined by BSAFE 1.x.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_BSAFE1_ENCRYPTION_PARAMS` structure:

```
typedef struct {
  int encryptionType;                                  /* encryption type */
} B_BSAFE1_ENCRYPTION_PARAMS;
```

`encryptionType` should be set to `B_BSAFE1_PAD` for pad mode, `B_BSAFE1_PAD_CHECKSUM` for pad with checksum mode, or `B_BSAFE1_RAW` for raw mode.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `B_BSAFE1_ENCRYPTION_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. Note that for pad mode or pad with checksum mode, `B_EncryptUpdate`, and `B_EncryptFinal` require a random algorithm. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for the *`randomAlgorithm`* argument in `B_DecryptUpdate` and `B_DecryptFinal`.

Notes:

If the input and output buffers of the RSA operation are interpreted as integers, the BSAFE 1.x format puts the least significant byte of the integer at the beginning of the buffer. This is in reverse order from algorithms in BSAFE 2.1 and later, such as `AI_RSAPrivate` and `AI_PKCS_RSAPrivate`, which put the most significant byte of the integer at the beginning of the buffer.

Algorithm methods to include in application's algorithm chooser:

`AM_RSA_CRT_ENCRYPT` or `AM_RSA_CRT_ENCRYPT_BLIND` for encryption, or `AM_RSA_CRT_DECRYPT` or `AM_RSA_CRT_DECRYPT_BLIND` for decryption.

`AM_RSA_CRT_ENCRYPT_BLIND` and `AM_RSA_CRT_DECRYPT_BLIND` perform blinding to protect against timing attacks, whereas `AM_RSA_CRT_ENCRYPT` and `AM_RSA_CRT_DECRYPT` do not.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_RSA_CRT`, `KI_PKCS_RSAPrivate`, `KI_PKCS_RSAPrivateBER`, or `KI_RSAPrivateBSAFE1`.

Input constraints:

In raw mode, the total number of input bytes must be a multiple of the key's modulus size in bytes. Also, each modulus-size block of input, interpreted as an integer with the most significant byte first, must be numerically less than the key's modulus.

Output considerations:

In pad mode and in pad with checksum mode, the output can be as large as one block greater than the number of input blocks, where each block is the size of the key's modulus size in bytes. For instance, given a 512-bit (64-byte) key and input of 64 bytes or less (one block), the output can be either 64 or 128 bytes long. With input of more than 64 bytes but less than or equal to 128 (two blocks), the output can be either 128 or 192 bytes long.

AI_RSAPublic

Purpose:

This AI allows you to decrypt data using the raw RSA algorithm. You can find the description of this algorithm in B. Schneier's *Applied Cryptography*.

`AI_RSAPublic` is different from `AI_PKCS_RSAPublic` because the latter allows you to decrypt *k*-11 bytes, where *k* is the size of the modulus in bytes, while you can use the former to decrypt up to *k* bytes. Note that it is the application's responsibility to strip the padding that was appended by the application to the data during encryption with `AI_RSAPrivate`.

Because this algorithm does not pad, the total number of input bytes must be a multiple of the key's modulus size in bytes. Also, each modulus-size block of input, interpreted as an integer with the most significant byte first, must be numerically less than the key's modulus.

To perform RSA decryption you can also use `AI_PKCS_RSAPublic` and `AI_SET_OAEP_RSAPublic`. But you can use `AI_RSAPublic` only if the data has been encrypted with `AI_RSAPrivate`.

Type of information this allows you to use:

the RSA algorithm for performing raw public-key decryption.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_RSA_ENCRYPT for encryption or AM_RSA_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_RSAPublic, KI_RSAPublicBER, or KI_RSAPublicBSAFE1.

Input constraints:

Because this algorithm does not pad, the total number of input bytes must be a multiple of the key's modulus size in bytes. Also, each modulus-size block of input, interpreted as an integer with the most significant byte first, must be numerically less than the key's modulus.

Token-based algorithm methods:

AI_RSAPublic may include the hardware algorithm methods AM_TOKEN_RSA_ENCRYPT, AM_TOKEN_RSA_DECRYPT, and AM_TOKEN_RSA_PUB_DECRYPT in the algorithm chooser for use with BHAPI.

Token-based key info types:

When used with one of the hardware algorithm methods described, AI_RSAPublic should be used with KI_Token or KI_KeypairToken.

AI_RSAPublicBSAFE1

Purpose:

Deprecated. This AI is included only for backward compatibility.

Type of information this allows you to use:

the decryption type parameter (pad, pad with checksum, or raw) for performing RSA public-key decryption as defined by BSAFE 1.x.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a B_BSAFE1_ENCRYPTION_PARAMS structure:

```
typedef struct {
  int encryptionType;                                /* encryption type */
} B_BSAFE1_ENCRYPTION_PARAMS;
```

encryptionType should be set to B_BSAFE1_PAD for pad mode, B_BSAFE1_PAD_CHECKSUM for pad with checksum mode, or B_BSAFE1_RAW for raw mode.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a B_BSAFE1_ENCRYPTION_PARAMS structure (see above).

Crypto-C procedures to use with algorithm object:

B_EncryptInit, B_EncryptUpdate, B_EncryptFinal, B_DecryptInit, B_DecryptUpdate, and B_DecryptFinal. Note that for pad mode or pad with checksum mode, B_EncryptUpdate and B_EncryptFinal require a random algorithm. You may pass (B_ALGORITHM_OBJ)NULL_PTR for the *randomAlgorithm* argument in B_DecryptUpdate and B_DecryptFinal.

Notes:

If the input and output buffers of the RSA operation are interpreted as integers, the BSAFE 1.x format puts the least significant byte of the integer at the beginning of the buffer. This is in reverse order from algorithms in BSAFE 2.1 and later, such as AI_RSAPublic and AI_PKCS_RSAPublic, which put the most significant byte of the integer at the beginning of the buffer.

Algorithm methods to include in application's algorithm chooser:

`AM_RSA_ENCRYPT` for encryption or `AM_RSA_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_RSAPublic`, `KI_RSAPublicBER`, or `KI_RSAPublicBSAFE1`.

Input constraints:

In raw mode, the total number of input bytes must be a multiple of the key's modulus size in bytes. Also, each modulus-size block of input, interpreted as an integer with the most significant byte first, must be numerically less than the key's modulus.

Output considerations:

In pad mode and in pad with checksum mode, the output can be as large as one block greater than the number of input blocks, where each block is the size of the key's modulus size in bytes. For instance, given a 512-bit (64-byte) key and input of 64 bytes or less (one block), the output can be either 64 or 128 bytes long. With input of more than 64 bytes but less than or equal to 128 (two blocks), the output can be either 128 or 192 bytes long.

AI_RSAStrongKeyGen

Purpose:

This AI allows you to specify the parameters for generating an RSA public/private key pair as defined in PKCS #1. The moduli generated are in conformance with the strength criteria of X9.31. If you do not want this level of conformance, you may use a faster `AI_RSAKeyGen` to generate RSA key pairs.

Type of information this allows you to use:

the parameters for generating an RSA public/private key pair as defined in PKCS #1, where the modulus size and public exponent are specified. The moduli generated are in conformance with the strength criteria of X9.31.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `A_RSA_KEY_GEN_PARAMS` structure:

```
typedef struct {
  unsigned int modulusBits;                    /* size of modulus in bits */
  ITEM               publicExponent;             /* fixed public exponent */
} A_RSA_KEY_GEN_PARAMS;
```

The `publicExponent` `ITEM` supplies an integer in canonical format, where the `ITEM`'s `data` points to an unsigned byte array, most significant byte first, and the `ITEM`'s `len` gives its length. All leading zeros are stripped from the integer before it is copied to the algorithm object. `modulusBits` must be at least 512 and must be a multiple of 16.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `A_RSA_KEY_GEN_PARAMS` structure (see above). All leading zeros have been stripped from the `publicExponent` integer.

Crypto-C procedures to use with algorithm object:

`B_GenerateInit` and `B_GenerateKeypair`. `B_GenerateKeypair` sets the `publicKey` key object with the `KI_RSAPublic` information and the `privateKey` key object with the `KI_PKCS_RSAPrivate` information. You must pass an initialized random algorithm to `B_GenerateKeypair`.

Algorithm methods to include in application’s algorithm chooser:

`AM_RSA_STRONG_KEY_GEN`.

AI_SET_OAEP_RSAPrivate

Purpose:

This AI allows you to decrypt data encrypted using `AI_SET_OAEP_RSAPublic`. This algorithm is used by the Secure Electronic Transaction (SET) protocol defined by Visa and MasterCard in the SET 1.0 specification released August 1, 1996. It replaces PKCS #1 v1.5 padding with a form of Optimal Asymmetric Encryption Padding (OAEP) that was developed for the SET protocol. OAEP provides protection against cryptanalytic attacks on the padding algorithm which are possible when most of the message being encrypted is known to the attacker. A more standard form of OAEP is now part of version 2.0 of the PKCS #1 standard and is implemented by `AI_PKCS_OAEP_RSAPrivate` and `AI_PKCS_OAEP_RSAPublic`.

Type of information this allows you to use:

the RSA algorithm for performing private-key decryption following the OAEP procedure outlined in the Aug. 1, 1996 version of the SET specifications.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, and `B_EncryptFinal` for encryption, and `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal` for decryption. `B_EncryptUpdate` and `B_EncryptFinal` require a random algorithm. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for the *`randomAlgorithm`* argument in `B_DecryptUpdate` and `B_DecryptFinal`.

Algorithm methods to include in application's algorithm chooser:

`AM_RSA_CRT_ENCRYPT` or `AM_RSA_CRT_ENCRYPT_BLIND` for encryption and `AM_RSA_CRT_DECRYPT` or `AM_RSA_CRT_DECRYPT_BLIND` for decryption. `AM_RSA_CRT_ENCRYPT_BLIND` and `AM_RSA_CRT_DECRYPT_BLIND` will perform blinding to protect against timing attacks, whereas `AM_RSA_CRT_ENCRYPT` and `AM_RSA_CRT_DECRYPT` do not.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_RSA_CRT`, `KI_PKCS_RSAPrivate`, `KI_PKCS_RSAPrivateBER`, or `KI_RSAPrivateBSAFE1`.

Input considerations:

The key size, in bits, must be a multiple of 8. For example, 1024 is a valid key size; 1030 is not.

If encrypting, the total number of bytes to encrypt must be 25 fewer than the key size in bytes. For instance, with a 1024-bit key (128 bytes) the input must be 103 bytes (128 - 25). The SET standard calls for the input data to follow a particular format. The first byte is the block content (BC) and the following bytes are the actual data bytes (ADB). This AI does not check whether those bytes adhere to the SET specifications.

Output considerations:

The output of encryption will be the same size as the key's modulus. The output of decryption will be 25 bytes fewer than the key size in bytes.

AI_SET_OAEP_RSAPublic

Purpose:

This AI allows you to encrypt data which will be decrypted using `AI_SET_OAEP_RSAPrivate`. This algorithm is used by the Secure Electronic Transaction (SET) protocol defined by Visa and MasterCard in the SET 1.0 specification released August 1, 1996. It replaces PKCS #1 v1.5 padding with a form of Optimal Asymmetric Encryption Padding (OAEP) that was developed for the SET protocol. OAEP provides protection against cryptanalytic attacks on the padding algorithm which are possible when most of the message being encrypted is known to the attacker. A more standard form of OAEP is now part of version 2.0 of the PKCS #1 standard and is implemented by `AI_PKCS_OAEP_RSAPrivate` and `AI_PKCS_OAEP_RSAPublic`.

Type of information this allows you to use:

the RSA algorithm for performing public-key encryption following the OAEP procedure outlined in the Aug. 1, 1996 version of the SET specifications.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. `B_EncryptUpdate` and `B_EncryptFinal` require a random algorithm. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for the *`randomAlgorithm`* argument in `B_DecryptUpdate` and `B_DecryptFinal`.

Algorithm methods to include in application's algorithm chooser:

`AM_RSA_ENCRYPT` for encryption and `AM_RSA_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_RSAPublic`, `KI_RSAPublicBER`, or `KI_RSAPublicBSAFE1`.

Input considerations:

The key size in bits must be a multiple of 8; for example, 1024 is a valid key size whereas 1030 is not.

If encrypting, the total number of bytes to encrypt must be 25 fewer than the key size in bytes. For instance, with a 1024-bit key (128 bytes) the input must be 103 bytes (128 - 25). The SET standard calls for the input data to follow a particular format. The first byte is the block content (BC) and the following bytes are the actual data bytes (ADB). This AI does not check whether those bytes adhere to the SET specifications.

Output considerations:

The output of encryption will be the same size as the key's modulus. The output of decryption will be 25 bytes fewer than the key size in bytes.

AI_SHA1

Purpose:

This AI allows you to create a message digest using the SHA1 digest algorithm as defined in FIPS PUB 180-1. This algorithm processes input data 64 bytes at a time but the length of the input does not have to be a multiple of 64 as the algorithm pads automatically.

The primary use for this AI is to authenticate data. Other algorithms that can be used for message digesting are `AI_MD2` and `AI_MD5` and their variants.

Type of information this allows you to use:

the 20-byte SHA1 message digest algorithm as defined in FIPS PUB 180-1.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_DigestInit`, `B_DigestUpdate`, and `B_DigestFinal`. Supply `NULL_PTR` for the *`keyObject`* argument in `B_DigestInit`.

Algorithm methods to include in application's algorithm chooser:

`AM_SHA`.

Compatible representation:

`AI_SHA1_BER`

Output considerations:

The output of `B_DigestFinal` will be 20 bytes long.

AI_SHA1_BER

Purpose:

This AI is similar to `AI_SHA1` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_SHA1` or `AI_SHA1_BER`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal is “43, 14, 3, 2, 26”.

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the SHA1 message digest algorithm as defined in FIPS PUB 180-1.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies a message digest algorithm other than SHA1.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_DigestInit`, `B_DigestUpdate`, and `B_DigestFinal`. Supply `NULL_PTR` for the *`keyObject`* argument in `B_DigestInit`.

Algorithm methods to include in application’s algorithm chooser:

`AM_SHA`.

Compatible representation:

`AI_SHA1`

Output considerations:

The output of `B_DigestFinal` will be 20 bytes long.

AI_SHA1Random

Purpose:

This AI allows you to generate a stream of pseudo-random numbers which are guaranteed to have a very high degree of randomness. Random numbers are used in deriving public and private keys, initialization vectors, and so on. This AI uses SHA1 as an underlying hashing function. The details of this algorithm are available from RSA Laboratories' Bulletin #8 or online at `http://www.rsasecurity.com/rsalabs/bulletins/`.

Other algorithms that can be used to generate pseudo-random numbers are `AI_MD2Random`, `AI_MD5Random`, and `AI_X962Random_V0.`

Notes:

In this API, `AI_SHA1Random` is identical to `AI_X962Random_V0` (Section 2.97); however, this identification may change in future versions of Crypto-C. For forward compatibility, we recommend that you do not use the name `AI_SHA1Random` in your applications; use `AI_X962Random_V0` instead.

`AI_X962Random_V0` provides an implementation of SHA1 Random that is based on the X9.62 standard; this is different from the implementation of SHA1 Random in RSA Security Inc.'s Java cryptographic toolkit, RSA BSAFE Crypto-J . Future versions of Crypto-C may implement `AI_SHA1Random` in a manner compatible with the implementation provided via the “SHA1Random” value passed to the `JSAFE_SecureRandom` class in Crypto-J.

AI_SHA1WithDES_CBCPad

Purpose:

This AI allows you to perform password-based encryption. This means that the input data will be encrypted with a secret key derived from a password, and it can be successfully decrypted only when the correct password is provided. Although this AI can be used to encrypt arbitrary data, its intended primary use is for encrypting private keys when transferring them from one computer system to another, as described in PKCS #8.

This AI employs DES secret-key encryption in cipher-block chaining (CBC) mode with padding, where the secret key is derived from a password using the SHA1 message digest algorithm. The details of this algorithm are contained in PKCS #5. DES is defined in FIPS PUB 81, and CBC mode of DES is defined in FIPS PUB 46-1. FIPS PUB 180-1 describes SHA1.

Other algorithms that can be used for password-based encryption are `AI_MD2WithDES_CBCPad`, `AI_MD2WithRC2_CBCPad`, `AI_MD5WithRC2_CBCPad`, and `AI_MD5WithDES_CBCPad`.

Type of information this allows you to use:

the salt and iteration count for the SHA1 With DES-CBC password-based encryption algorithm. The salt is concatenated with the password before being digested by SHA1, and the iteration count specifies how many times the digest needs to be run. The count of 2 indicates that the result of digesting the password-and-salt string needs to be run once more through SHA1. The first 8 bytes of the final digest become the secret key for the DES cipher after being adjusted for parity as required by FIPS PUB 81, the next 8 bytes become the initialization vector, and the last 4 bytes are ignored.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to a `B_PBE_PARAMS` structure:

```
typedef struct {
  unsigned char *salt;                     /* pointer to 8-byte salt value */
  unsigned int   iterationCount;                         /* iteration count */
} B_PBE_PARAMS;
```

RSA Security Inc. recommends a minimum iteration count of 1,000. However, for an additional byte or two of security the iteration should be 2^8 to 2^{16}.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to a `B_PBE_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, and `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_SHA` and `AM_DES_CBC_ENCRYPT` for encryption or `AM_DES_CBC_DECRYPT` for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

`KI_Item` that gives the address and length of the password.

Compatible representation:

`AI_SHA1WithDES_CBCPadBER`.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_SHA1WithDES_CBCPadBER

Purpose:

This AI is similar to `AI_SHA1WithDES_CBCPad` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier which includes ASN.1 encoding of the `B_PBE_PARAMS` structure defined in the description of `AI_SHA1WithDES_CBCPad`. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_SHA1WithDES_CBCPad` or `AI_SHA1WithDES_CBCPad`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal is "`42, 134, 72, 134, 247, 13, 1, 5, 10`".

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies the SHA1 With DES-CBC password-based encryption algorithm, as defined by RSA Security Inc.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than SHA1 With DES-CBC.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`, `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_SHA and AM_DES_CBC_ENCRYPT for encryption or AM_DES_CBC_DECRYPT for decryption.

Key info types for *keyObject* in B_EncryptInit or B_DecryptInit:

KI_Item that gives the address and length of the password.

Compatible representation:

AI_SHA1WithDES_CBCPad.

Output considerations:

During encryption, this AI pads the output (which DES requires to be a multiple of 8 bytes long). Thus, the total number of output bytes from encryption can be as many as 8 bytes more than the total input.

AI_SHA1WithRSAEncryption

Purpose:

This AI allows you to perform signature operations that involve the SHA1 digest algorithm and RSA public key algorithm. The digest of a message is created using the SHA1 algorithm, and then it is signed using PKCS#1 digital signature algorithm. Other algorithms that can be used for the same purpose are `AI_MD2WithRSAEncryption` and `AI_MD5WithRSAEncryption`.

Type of information this allows you to use:

RSA Security Inc.'s SHA1 With RSA signature algorithm that uses the SHA1 digest algorithm and the RSA algorithm to create and verify RSA digital signatures as defined in PKCS #1.

Note that in order to perform PKCS #1 digital signatures with a 20-byte digest, the RSA key must be at least 368 bits long.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`.

Format of *info* returned by B_GetAlgorithmInfo:

`NULL_PTR`.

Crypto-C procedures to use with algorithm object:

`B_SignInit`, `B_SignUpdate`, `B_SignFinal`, `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_MD2`, and `AM_RSA_CRT_ENCRYPT`, `AM_RSA_CRT_ENCRYPT_BLIND`, or `AM_RSA_ENCRYPT`, for signature creation; and `AM_RSA_DECRYPT` for signature verification. `AM_RSA_CRT_ENCRYPT_BLIND` performs blinding to protect against timing attacks, whereas `AM_RSA_CRT_ENCRYPT` does not.

Key info types for *keyObject* in B_SignInit:

`KI_RSA_CRT`, `KI_PKCS_RSAPrivate`, `KI_PKCS_RSAPrivateBER`, `KI_RSAPrivate` or `KI_RSAPrivateBSAFE1`. Unless you use `KI_RSA_CRT` for your KI, you must include `AM_RSA_ENCRYPT` in your application's algorithm chooser.

Key info types for *keyObject* in B_VerifyInit:

`KI_RSAPublic`, `KI_RSAPublicBER` or `KI_RSAPublicBSAFE1`.

Compatible representation:

`AI_SHA1WithRSAEncryptionBER`.

Output considerations:

The *`signature`* result of `B_SignFinal` will be the same size as the RSA key's modulus.

Notes:

Although the RSA signature operation is called "encryption" and the verification operation is called "decryption", the signer uses the digest and the private key and follows the steps needed to decrypt, while the verifier uses the transmitted digest and the public key and follows the steps needed to encrypt.

AI_SHA1WithRSAEncryptionBER

Purpose:

This AI is similar to `AI_SHA1WithRSAEncryption` except that it uses the ASN.1 BER format. This AI allows you to parse and create ASN.1 algorithm identifiers such as those used in PKCS #7 and other protocols. You call `B_SetAlgorithmInfo` to initialize an algorithm object from the encoded algorithm identifier. You call `B_GetAlgorithmInfo` with this AI to create an encoded algorithm identifier from an algorithm object that was created using `AI_SHA1WithRSAEncryption` or `AI_SHA1WithRSAEncryptionBER`. The `OID` for this algorithm, excluding the tag and length bytes, in decimal is “`42, 134, 72, 134, 247, 13, 1, 1, 5`”.

Type of information this allows you to use:

the encoding of an algorithm identifier that specifies RSA Security Inc.'s SHA1 With RSA signature algorithm that uses the SHA1 digest algorithm and the RSA algorithm to create and verify RSA digital signatures as defined in PKCS #1.

Note that in order to perform PKCS #1 digital signatures with a 20-byte digest, the RSA key must be at least 368 bits long.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the BER-encoded algorithm identifier. The encoding is converted to DER before it is copied to the algorithm object. `B_SetAlgorithmInfo` returns `BE_WRONG_ALGORITHM_INFO` if the algorithm identifier specifies an algorithm other than SHA1 With RSA Encryption.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an `ITEM` structure that gives the address and length of the DER-encoded algorithm identifier.

Crypto-C procedures to use with algorithm object:

`B_SignInit`, `B_SignUpdate`, `B_SignFinal`, `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal`. You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

AM_MD2, and AM_RSA_CRT_ENCRYPT, AM_RSA_CRT_ENCRYPT_BLIND, or AM_RSA_ENCRYPT, for signature creation; and AM_RSA_DECRYPT for signature verification. AM_RSA_CRT_ENCRYPT_BLIND performs blinding to protect against timing attacks, whereas AM_RSA_CRT_ENCRYPT does not.

Key info types for *keyObject* in B_SignInit:

KI_RSA_CRT, KI_PKCS_RSAPrivate, KI_PKCS_RSAPrivateBER, KI_RSAPrivate or KI_RSAPrivateBSAFE1. Unless you use KI_RSA_CRT for your KI, you must include AM_RSA_ENCRYPT in your application's algorithm chooser.

Key info types for *keyObject* in B_VerifyInit:

KI_RSAPublic, KI_RSAPublicBER, or KI_RSAPublicBSAFE1.

Compatible representation:

AI_SHA1WithRSAEncryption.

Output considerations:

The *signature* result of B_SignFinal will be the same size as the RSA key's modulus.

Notes:

Although the RSA signature operation is called "encryption" and the verification operation is called "decryption", the signer uses the digest and the private key and follows the steps needed to decrypt, while the verifier uses the transmitted digest and the public key and follows the steps needed to encrypt.

AI_SignVerify

Purpose:

This AI allows you to sign a message in compliance with the X9.31 standard, or to verify X9.31 compliant signatures.

Type of information this allows you to use:

an RSA private key to sign a message in compliance with the X9.31 standard, or an RSA public key to verify X9.31 compliant signatures. To be fully compliant with the X9.31 standard, use only keys that are generated by `AI_RSAStrongKeyGen`.

Format of *info* supplied to B_SetAlgorithmInfo:

a pointer to a `B_SIGN_VERIFY_PARAMS` structure:

```
typedef struct {                                         /* Current Choices */
  unsigned char *encryptionMethodName;   /* “rsaSignX931”, “rsaVerifyX931” */
  POINTER       encryptionParams;  /* Null for what is currently available*/
  unsigned char *digestMethodName;                                /* “sha1” */
  POINTER       digestParams;                             /* Null for sha1 */
  unsigned char *formatMethodName;                          /*”formatX931” */
  POINTER       formatParams; /* structure of type A_X931_PARAMS for sha1 */
} B_SIGN_VERIFY_PARAMS;
```

Set the arguments for `B_SIGN_VERIFY_PARAMS` as follows:

Argument	Values	Comments
encryptionMethodName	`rsaSignX931`	
	`rsaVerifyX931`	
encryptionParams	`Null Pointer for RSA X9.31`	
digestMethodName	`sha1`	
digestParams	`Null Pointer for SHA1`	

Argument	Values	Comments
formatMethodName	`formatX931`	Signature is output as the binary concatenation of r with s.
	`formatX931PKCS`	Signature is output in PKCS #1 style DER encoding.
	`formatX931X957`	Signature is output in X9.57 (DSA) DER format.
formatParams	`Pointer to structure of type A_X931_PARAMS`	Use with rsaSignX931 and rsaVerifyX931.

If *formatMethodName* is "`formatX931`", *formatParams* must be given a pointer to a structure of type `A_X931_PARAMS`:

```
typedef struct {
  unsigned int blockLen;
  unsigned int oidNum;
  ITEM OID;
} A_X931_PARAMS;
```

Set the arguments for `A_X931_PARAMS` as follows:

Argument	Values	Comments
blockLen	`Number of bytes in final X.931 signature`	This is the number of bytes that the binary concatenation of r with s requires.
oidNum	`The object identifier for the SHA1 hash algorithm`	Currently always equals 3 (SHA) for X9.31.
OID	`Object identifier to use for BER encoding`	Currently only used when "formatX931PKCS" is specified in the ***formatMethodName***. PKCS #1 style BER/DER encoding is used.

Format of *info* returned by B_GetAlgorithmInfo:

a pointer to a `B_SIGN_VERIFY_PARAMS` structure (see above).

Crypto-C procedures to use with algorithm object:

`B_SignInit`, `B_SignUpdate`, `B_SignFinal`, `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal`. You may pass (`B_ALGORITHM_OBJ`)`NULL_PTR` for all *`randomAlgorithm`* arguments.

Algorithm methods to include in application's algorithm chooser:

`AM_SHA`, `AM_FORMAT_X931`, and `AM_RSA_CRT_X931_ENCRYPT` for signature creation; or `AM_SHA`, `AM_EXTRACT_X931`, and `AM_RSA_X931_DECRYPT` for signature verification.

Key info types for *keyObject* in B_SignInit or B_VerifyInit:

`KI_RSAPrivate` or compatible key types to create a signature, or `KI_RSAPublic` or compatible key types to verify a signature.

AI_SymKeyTokenGen

Purpose:

This AI allows you to generate the token form of a symmetric-cipher key.

Type of information this allows you to use:

the parameters for generating the token form of a symmetric key. The BSAFE Hardware API (BHAPI) supports token forms of DES, RC2, RC4, RC5, and Triple DES keys.

Format of *info* supplied to B_SetAlgorithmInfo:

pointer to an A_SYMMETRIC_KEY_SPECIFIER structure:

```
typedef struct {
  unsigned int   keyUsage;                      /* X509 key usage bit map */
  unsigned int   keyLengthInBytes;
  unsigned long  lifeTime;            /* Key lifetime; under consideration */
  unsigned int   protectFlag;               /* Store key in encrypted form */
  unsigned char *cipherName;          /* String tag for key's cipher class */
                                                         /*  , e.g., “des” */
} A_SYMMETRIC_KEY_SPECIFIER;
```

where `cipherName` is one of: “`des`”, “`desx`”, “`rc2`”, “`rc4`”, “`rc5`”, or “`triplexes`”.

Format of *info* returned by B_GetAlgorithmInfo:

pointer to an A_SYMMETRIC_KEY_SPECIFIER structure.

Crypto-C procedures to use with algorithm object:

`B_SymmetricKeyGenerateInit` and `B_SymmetricKeyGenerate`.

Algorithm methods to include in application’s algorithm chooser:

AM_SYMMETRIC_KEY_TOKEN_GEN

Notes:

Can only be used in conjunction with a hardware implementation; if no hardware implementation is present, `AI_SymKeyTokenGen` does not do anything. `AI_SymKeyTokenGen` can only be used if you have called `B_CreateSessionChooser` for your application.

The corresponding software-based method is a `HW_TABLE_ENTRY SF_SYMMETRIC_KEY_TOKEN_GEN`. This provides software support in the case that hardware is unavailable. This method can be utilized only by including the software chooser inside the hardware chooser table.

AI_X931Random

Purpose:

This AI allows you to generate random numbers for RSA key generation in conformance with X9.31. This AI can be used to supply multiple independent streams of randomness. It is included in Crypto-C mainly to satisfy the requirements of independent generation of large and private prime factors, as specified by X9.31.

This AI is intended for use with `AI_RSAStrongKeyGen` only. The SHA1 based pseudo-random number generator, *G(sha1)*, which is defined in the X9.31 standard and in the FIPS182-1 DSA standard, is available as `AI_X962Random_V0`. If you are not using X9.31 rDSA signatures but require the *G(sha1)* hash function, you should use `AI_X962Random_V0` and not `AI_X931Random`.

Type of information this allows you to use:

A SHA1 based pseudo-random number generator as defined in Appendix A of the X9.31 standard.

Format of info supplied to B_SetAlgorithmInfo:

`NULL_PTR`, if there is only one stream and no additional seeding is desired,

or

a pointer to an `A_X931_RANDOM_PARAMS` structure:

```
typedef struct
{
     unsigned int numberOfStreams;        /* number of independent streams */
     ITEM         seed;                                /* additional seeding */
                                  /* to be equally divided among the streams */
} A_X931_RANDOM_PARAMS;
```

When `AI_X931Random` is used with `AI_RSAStrongKeyGen`, the *`numberOfStreams`* field must always be equal to 6.

Format of info returned by B_GetAlgorithmInfo:

returns a `NULL_PTR` if set with `NULL_PTR`; returns a pointer to an

A_X931_RANDOM_PARAMS structure otherwise.

BSAFE procedures to use with algorithm object:

B_RandomInit, B_RandomUpdate, and B_GenerateRandomBytes, and as the *randomAlgorithm* argument to other procedures.

Algorithm methods to include in application's algorithm chooser:

AM_X931_RANDOM.

Notes

This AI is intended for use with AI_StrongKeyGen. When used with AI_StrongKeyGen, *numberOfStreams* must always be equal to 6.

Internal to this implementation, for *numberOfStreams* = 6, the entropy is divided as follows (for the purpose of this explanation, bits of entropy are bits that are unpredictable):

- Seeds 1, 2, 4, and 5 need only 101 bits, or 13 bytes, of entropy.
- Seeds 3 and 6 need one-half times the modulus length of entropy. Therefore, for example, for a modulus of 1024 bits, 64 bytes of entropy are necessary. For a modulus size of 1536 bits, 92 bytes of entropy are necessary, and for a modulus size of 2048 bits, 128 bytes of entropy are necessary.

Although seeds 1, 2, 4, and 5 only need 13 bytes (101 bits) of random seed, all 6 of the seeds take up the same amount of physical space in memory. For example, for the modulus size of 1536 bits, you will seed a buffer of 6 times 92 bytes in length, or 552 bytes. This buffer is then divided into 6 evenly distributed 92-byte fields. For fields 1, 2, 4, and 5 you only need to worry about placing 13 bytes of random seed information into them. For fields 3 and 6 you would need to fill all 92 bytes of memory with random seed data.

Although Crypto-C does some basic error checking, the quality of the application's entropy is not measured fully by AI_X931Random, and it may be that no error is returned even if seed entropy is poor or if insufficient random streams are provided. The proper implementation of sufficient entropy sources is *the responsibility of the application*, and *not* of Crypto-C. If a hardware random number generator, such as the Intel Random Number Generator, is available, it would be a good source for a random seed. See the *Intel Security Hardware User's Guide* for more information on using Crypto-C with the Intel RNG.

AI_X962Random_V0

Purpose:

This AI allows you to generate a stream of pseudo-random numbers which are guaranteed to have a very high degree of randomness. Random numbers are used in deriving public and private keys, initialization vectors, etc. This AI uses SHA1 as an underlying hashing function. The details of this algorithm are specified in X9.62, and it is similar to the algorithm in section A.2.1 of X9.31.

This algorithm can produce numbers between zero and the value of a given prime minus one. Such numbers are useful for the U.S. Government Digital Signature Standard (DSS).

Other algorithms that can be used to generate pseudo-random numbers are `AI_MD2Random`, `AI_SHA1Random`, and `AI_MD5Random`.

Type of information this allows you to use:

the SHA1 pseudo-random generator as defined in X9.62.

Format of *info* supplied to B_SetAlgorithmInfo:

`NULL_PTR`, if it is desired to use the `AI_X962Random_V0` object in the same fashion as `AI_MD5Random`.

a pointer to an `A_SHA_RANDOM_PARAMS` structure:.

```
typedef struct {
ITEM prime;               /* Optional input for X-9.62 mode only. Used to */
                      /* generate a pseudo-random number (but not uniform) */
                     /* in [1, prime - 1].  Set prime.len to zero otherwise */
ITEM seed;            /* Special additional seeding of 20 to 128 bytes long.*/
            /* May be used in place of usual B_UpdateRandom seeding calls, */
        /* but requires the availability of nearly perfectly random bytes. */
                           /* If B_UpdateRandom seeding calls are used, then */
                 /* this additional seeding material is used to augment the */
                      /* randomness of the pseudo-random numbers generated.*/
} A_SHA_RANDOM_PARAMS;
```

Format of *info* returned by B_GetAlgorithmInfo:

returns a NULL_PTR if set with NULL_PTR; otherwise, returns a pointer to an A_SHA_RANDOM_PARAMS structure.

Crypto-C procedures to use with algorithm object:

B_RandomInit, B_RandomUpdate, and B_GenerateRandomBytes, and as the *randomAlgorithm* argument to other procedures.

Algorithm methods to include in application's algorithm chooser:

AM_SHA_RANDOM.

Notes:

There are a number of possible implementations of SHA1 pseudo-random number generation. AI_X962Random_V0 implements a SHA1 Random generator that is based on X9.62. The FIPS 186 standard defines a similar algorithm (also defined in X9.31), but due to slight differences between FIPS 186 and X9.62, the same seeding sequence will produce different outputs. In addition, AI_X962Random_V0's implementation of SHA1 Random is substantially different from the implementation in RSA Security Inc.'s Java cryptographic toolkit, Crypto-J.

Chapter 3

Key Info Types

This chapter lists the standard key information types (KIs) offered with Crypto-C. For an explanation of the key object, see "The Key Object" on page 9. A typical application supplies a key info type as the *`infoType`* argument to `B_SetKeyInfo`. For examples of how to use key info types with certain algorithms, see the *User's Manual*.

The format for each entry is shown in Figure 3-1 on page 246.

Purpose:

Describes the KI, what it is for, and what it does.

Type of information this allows you to use:

Describes the type and format of key information you can use with the key info type.

Format of *info* returned by B_GetKeyInfo:

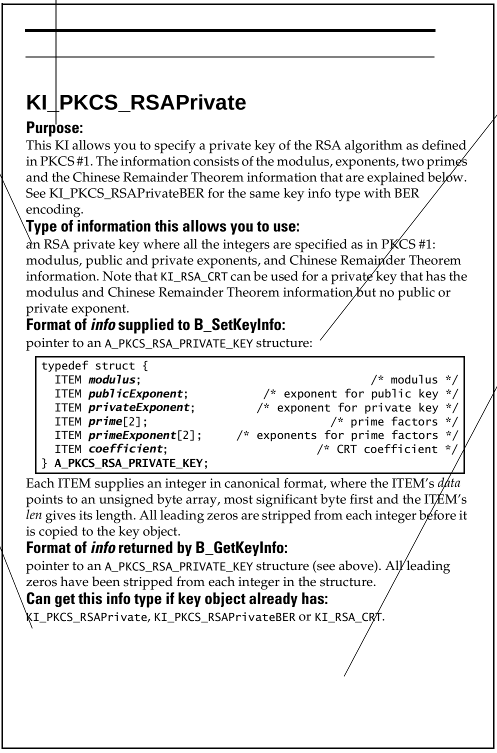
Describes the exact format that B_GetKeyInfo returns for the key value. This is generally a “cleaned up” version of the format supplied to B_SetKeyInfo. For example, B_GetKeyInfo with KI_DES8 returns the DES key with the DES key parity set.

Format of *info* supplied to B_SetKeyInfo:

Describes the exact format for supplying the key value to B_SetKeyInfo.

Can get this info type if key object already has:

Most keys have multiple representations for the key information. For example, you can specify an 8-byte RC2 key with KI_8Byte or KI_Item. This describes what type of key information a key object must already have if you want to call B_GetKeyInfo using this key info type.



KI_PKCS_RSAPrivate

Purpose:

This KI allows you to specify a private key of the RSA algorithm as defined in PKCS #1. The information consists of the modulus, exponents, two primes and the Chinese Remainder Theorem information that are explained below. See KI_PKCS_RSAPrivateBER for the same key info type with BER encoding.

Type of information this allows you to use:

an RSA private key where all the integers are specified as in PKCS #1: modulus, public and private exponents, and Chinese Remainder Theorem information. Note that KI_RSA_CRT can be used for a private key that has the modulus and Chinese Remainder Theorem information but no public or private exponent.

Format of *info* supplied to B_SetKeyInfo:

pointer to an A_PKCS_RSA_PRIVATE_KEY structure:

```
typedef struct {
  ITEM modulus;                                  /* modulus */
  ITEM publicExponent;          /* exponent for public key */
  ITEM privateExponent;        /* exponent for private key */
  ITEM prime[2];                           /* prime factors */
  ITEM primeExponent[2];     /* exponents for prime factors */
  ITEM coefficient;                    /* CRT coefficient */
} A_PKCS_RSA_PRIVATE_KEY;
```

Each ITEM supplies an integer in canonical format, where the ITEM’s *data* points to an unsigned byte array, most significant byte first and the ITEM’s *len* gives its length. All leading zeros are stripped from each integer before it is copied to the key object.

Format of *info* returned by B_GetKeyInfo:

pointer to an A_PKCS_RSA_PRIVATE_KEY structure (see above). All leading zeros have been stripped from each integer in the structure.

Can get this info type if key object already has:

KI_PKCS_RSAPrivate, KI_PKCS_RSAPrivateBER or KI_RSA_CRT.

Figure 3-1 **Sample Key Info Type**

KI_8Byte

Purpose:

This KI allows you to specify a generic 8-byte key for a symmetric encryption algorithm that may be RC2, DES, or any other symmetric algorithm. For DES encryption, there exist more specific variants of 8-byte key info types: `KI_DES8` and `KI_DES8Strong`.

Type of information this allows you to use:

an 8-byte value for symmetric keys such as DES and RC2. Note that `KI_DES8Strong` is usually used for a DES key because it sets the DES parity and checks for weak DES keys.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `unsigned char` array that holds the 8 bytes.

Format of *info* returned by B_GetKeyInfo:

pointer to an `unsigned char` array that holds the 8 bytes.

Can get this info type if key object already has:

`KI_8Byte`, `KI_Item` (if the length of the `ITEM` is 8), or `KI_DES8`.

KI_24Byte

Purpose:

This KI allows you to specify a generic 24-byte key for a symmetric encryption algorithm such as Triple DES. It may also be used as keying material for certain MAC algorithms such as HMAC. See `KI_DES24Strong` for a Triple DES specific variant of this key info type.

Type of information this allows you to use:

a 24-byte value for symmetric keys such as `DES_EDE`.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `unsigned char` array that holds the 24 bytes.

Format of *info* returned by B_GetKeyInfo:

pointer to an `unsigned char` array that holds the 24 bytes.

Can get this info type if key object already has:

`KI_24Byte`, `KI_Item` (if the length of the `ITEM` is 24), `KI_DESX`.

KI_DES8

Purpose:

This KI allows you to specify an 8-byte key used by the DES algorithm. The key object will satisfy the DES parity requirement. Unlike `KI_DES8Strong`, it does not check against known DES weak keys. See the section "DES Weak Keys" in the *User's Manual*.

Type of information this allows you to use:

an 8-byte value for a DES key where the information stored in the key object must be DES-parity adjusted according to FIPS 46-1. Crypto-C treats the least significant bit of each byte of the key data as the DES parity adjustment bit.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `unsigned char` array that holds the 8-byte DES key. The key is DES parity adjusted when it is copied to the key object.

For added security, it is prudent to check the proposed key data against known byte sequences that produce weak DES keys before calling `B_SetKeyInfo`. See the section "DES Weak Keys" in the *User's Manual*.

Format of *info* returned by B_GetKeyInfo:

pointer to an `unsigned char` array that holds the 8-byte DES key that is DES parity adjusted.

Can get this info type if key object already has:

`KI_DES8`, `KI_Item` (if the length of the `ITEM` is 8 and the data's DES parity is correct), or `KI_8Byte` (if the DES parity is correct).

Notes:

It is more secure to use `KI_DES8Strong` instead of `KI_DES8`. When you call `B_SetAlgorithmInfo` with `KI_DES8Strong`, Crypto-C checks the key against a list of known weak keys and returns an error if the resulting key would be weak. See the section "DES Weak Keys" in the *User's Manual*.

KI_DES8Strong

Purpose:

This KI allows you to specify an 8-byte key used by the DES algorithm. The key object will satisfy the DES parity requirement and will be checked against known DES weak keys. Also see `KI_DES8.`

Type of information this allows you to use:

an 8-byte value for a DES key where the information stored in the key object will be DES parity adjusted according to FIPS 46-1. Crypto-C treats the least significant bit of each byte of the key data as the DES parity adjustment bit. When setting a key object with this KI, Crypto-C will check the input data against a list of known DES weak keys. If the resulting key would be weak, Crypto-C returns an error.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `unsigned char` array that holds the 8-byte DES key. The key is DES parity adjusted when it is copied to the key object.

Format of *info* returned by B_GetKeyInfo:

pointer to an `unsigned char` array that holds the 8-byte DES key, which is DES parity adjusted.

Can get this info type if key object already has:

`KI_DES8Strong`, `KI_DES8` (if the key is not weak), `KI_Item` (if the length of the `ITEM` is 8, the data's DES parity is correct, and the key is not weak), or `KI_8Byte` (if the DES parity is correct and the key is not weak.

KI_DES24Strong

Purpose:

This KI allows you to specify a 24-byte key used by the Triple DES algorithm. The key object will satisfy the DES parity requirement and will be checked against known DES weak keys.

Type of information this allows you to use:

24-byte value for a Triple DES key where the information stored in the key object will be DES parity-adjusted according to FIPS 46-1. Crypto-C treats the least significant bit of each byte of the key data as the DES parity-adjustment bit. When setting a key object with this KI, Crypto-C will check the input data against a list of known DES weak keys. If the resulting key would be weak, Crypto-C returns an error.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `unsigned char` array that holds the 24-byte Triple DES key. The key is DES parity adjusted when it is copied to the key object.

Format of *info* returned by B_GetKeyInfo:

pointer to an `unsigned char` array that holds the 24-byte Triple DES key that is DES parity adjusted.

Can get this info type if key object already has:

`KI_DES24Strong`, `KI_24Byte` (if the key is not weak), `KI_Item` (if the length of the `ITEM` is 24 and the key is not weak), `KI_DESX` (if the key is not weak).

KI_DES_BSAFE1

Purpose:

Deprecated. This type is included only for backward compatibility.

Type of information this allows you to use:

the BSAFE 1.x encoding of a DES key.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BSAFE 1.x encoding. `B_SetKeyInfo` returns `BE_WRONG_KEY_INFO` if the encoding specifies a secret key for an algorithm other than DES.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BSAFE 1.x encoding.

Can get this info type if key object already has:

`KI_DES_BSAFE1`, `KI_DES8`, `KI_Item` (if the length of the `ITEM` is 8 and the data's DES parity is correct), `KI_8Byte` (if the DES parity is correct), `KI_RC2WithBSAFE1Params` (if the DES parity of the RC2 key is correct) or `KI_RC2_BSAFE1` (if the DES parity of the 8-byte RC2 key is correct).

KI_DESX

Purpose:

This KI allows you to specify keying materials for the DESX algorithm. They include the key value, input whitener, and output whitener. The key value will be used as the DES encryption key for the DESX algorithm and thus it will be parity adjusted. See the `AI_DESX_CBC_IV8` section for descriptions of the DESX algorithm.

Type of information this allows you to use:

a DESX key where the key value, input whitener, and output whitener are specified.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `A_DESX_KEY` structure:

```
typedef struct {
  unsigned char *key;                           /* pointer to 8-byte key */
  unsigned char *inputWhitener;      /* pointer to 8-byte input whitener */
  unsigned char *outputWhitener;    /* pointer to 8-byte output whitener */
} A_DESX_KEY;
```

The value of key is DES parity adjusted when it is copied to the key object.

Format of *info* returned by B_GetKeyInfo:

pointer to an `A_DESX_KEY` structure (see above). The value of key is DES parity adjusted.

Can get this info type if key object already has:

`KI_24Byte` (if the DES parity is correct), `KI_Item` (if the length of the `ITEM` is 24 and the data's DES parity is correct), or `KI_DESX`.

KI_DESX_BSAFE1

Purpose:

Deprecated. This type is included only for backward compatibility.

Type of information this allows you to use:

the BSAFE 1.x encoding of a DESX key.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BSAFE 1.x encoding. `B_SetKeyInfo` returns `BE_WRONG_KEY_INFO` if the encoding specifies a secret key for an algorithm other than DESX.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BSAFE 1.x encoding.

Can get this info type if key object already has:

`KI_DESX_BSAFE1` or `KI_DESX` (if the output whitener is the MAC of the key and the input whitener).

KI_DSAPrivate

Purpose:

This KI allows you to specify a private key used by the DSA algorithm. The information consists of a private component and the three parameters: *p*, *q*, and *g*, which are explained below. See `KI_DSAPrivateBER` or `KI_DSAPrivateX957BER` for the same key type with BER encoding.

Type of information this allows you to use:

a DSA private key. The parameters of the key are specified as the following: private component (*x*), the prime (*p*), the subprime (*q*) and the base (*g*).

Format of *info* supplied to B_SetKeyInfo:

pointer to an `A_DSA_PRIVATE_KEY` structure:

```
typedef struct {
  ITEM x;                                         /* private component */
  A_DSA_PARAMS params;                 /* the DSA parameters, p, q and g */
} A_DSA_PRIVATE_KEY;
```

where `A_DSA_PARAMS` is defined as

```
typedef struct {
  ITEM prime;                                            /* the prime p */
  ITEM subPrime;                                      /* the subprime q */
  ITEM base;                                              /* the base g */
} A_DSA_PARAMS;
```

Each `ITEM` supplies an integer in canonical format, where the `ITEM`'s data points to an unsigned byte array, most significant byte first, and the `ITEM`'s len gives its length. All leading zeros are stripped from each integer before it is copied to the key object.

Format of *info* returned by B_GetKeyInfo:

pointer to an `A_DSA_PRIVATE_KEY` structure (see above). All leading zeros have been stripped from each integer in the structure.

Can get this info type if key object already has:

`KI_DSAPrivate`, `KI_DSAPrivateBER`, or `KI_DSAPrivateX957BER`.

KI_DSAPrivateBER

Purpose:

This KI is similar to KI_DSAPrivate except that it uses the ASN.1 BER format. This KI allows you to parse and create an ASN.1 key info type encoded with the PKCS #8 standard. You call B_SetKeyInfo to initialize a key object from the encoded key info type that includes the private component, prime, subprime, and base. You call B_GetKeyInfo with this KI to create an encoded key info type from a key object that was created using KI_DSAPrivate or KI_DSAPrivateBER. The OID for DSA keys, excluding the tag and length bytes, in decimal, is "43, 14, 3, 2, 12". Also see KI_DSAPrivate and KI_DSAPrivateX957BER.

Type of information this allows you to use:

the encoding of a DSA private key that is encoded as a PKCS #8 PrivateKeyInfo type and that contains an RSA Security Inc. DSAPrivateKey type. Note that this encoding contains all of the information specified by KI_DSAPrivate.

Format of *info* supplied to B_SetKeyInfo:

pointer to an ITEM structure that gives the address and length of the BER encoding. The encoding is converted to DER before it is copied to the key object. B_SetKeyInfo returns BE_WRONG_KEY_INFO if the PrivateKeyInfo specifies a private key for an algorithm other than DSA.

Format of *info* returned by B_GetKeyInfo:

pointer to an ITEM structure that gives the address and length of the DER encoding.

Can get this info type if key object already has:

KI_DSAPrivate, KI_DSAPrivateBER, or KI_DSAPrivateX957BER.

KI_DSAPrivateX957BER

Purpose:

This KI represents the same algorithm type as `KI_DSAPrivate` except that it uses the ASN.1 BER format. It allows you to parse and create an ASN.1 key info type encoded as specified by X9.57. You call `B_SetKeyInfo` to initialize a key object from the encoded key info type that includes the private component, prime, subprime, and base. You call `B_GetKeyInfo` with this KI to create an encoded key info type from a key object that was created using `KI_DSAPrivate`, `KI_DSAPrivateBER`, or `KI_DSAPrivateX957BER`. The OID for DSA keys, excluding the tag and length bytes, in decimal, is “`42, 134, 72, 206, 56, 4, 1`”. Also see `KI_DSAPrivate` and `KI_DSAPrivateBER`.

Type of information this allows you to use:

the encoding of a DSA private key that is encoded as X9.57 PrivateKeyInfo type that contains an RSA Security Inc. DSAPrivateKey type. Note that this encoding contains all of the information specified by `KI_DSAPrivate`.

Format of info supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BER encoding. The encoding is converted to DER before it is copied to the key object. `B_SetKeyInfo` returns `BE_WRONG_KEY_INFO` if the PrivateKeyInfo specifies a private key for an algorithm other than DSA (as defined by X9.57).

Format of info returned by B_GetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the DER encoding.

Can get this info type if key object already has:

`KI_DSAPrivate`, `KI_DSAPrivateBER`, or `KI_DSAPrivateX957BER`.

KI_DSAPublic

Purpose:

This KI allows you to specify a public key used by the DSA algorithm. The information consists of a private component and the three parameters: *p*, *q*, and *g*, which are explained below. See KI_DSAPublicBER or KI_DSAPublicX957BER for the same key type with BER encoding.

Type of information this allows you to use:

a DSA public key where all the parameters are specified as in X9.30 Part III: the public component (*y*), the prime (*p*), the subprime (*q*), and the base (*g*).

Format of *info* supplied to B_SetKeyInfo:

pointer to an A_DSA_PUBLIC_KEY structure:

```
typedef struct {
  ITEM y;                                            /* public component */
  A_DSA_PARAMS params;                   /* the DSA parameters, p, q and g */
} A_DSA_PUBLIC_KEY;
```

where A_DSA_PARAMS is defined as:

```
typedef struct {
  ITEM prime;                                               /* the prime p */
  ITEM subPrime;                                         /* the subprime q */
  ITEM base;                                                 /* the base g */
} A_DSA_PARAMS;
```

Each ITEM supplies an integer in canonical format, where the ITEM's *data* points to an unsigned byte array, most significant byte first, and the ITEM's *len* gives its length. All leading zeros are stripped from each integer before it is copied to the key object.

Format of *info* returned by B_GetKeyInfo:

pointer to an A_DSA_PUBLIC_KEY structure (see above). All leading zeros have been stripped from each integer in the structure.

Can get this info type if key object already has:

`KI_DSAPublic`, `KI_DSAPublicBER`, `KI_DSAPublicX957BER`.

KI_DSAPublicBER

Purpose:

This KI is similar to `KI_DSAPublic` except that it uses the ASN.1 BER format. This KI allows you to parse and create an ASN.1 key info type encoded with the X.509 standard for SubjectPublicKeyInfo. You call `B_SetKeyInfo` to initialize a key object from the encoded key info type that includes the public component, prime, subprime, and base. You call `B_GetKeyInfo` with this KI to create an encoded key info type from a key object that was created using `KI_DSAPublic` or `KI_DSAPublicBER`. The `OID` for DSA keys, excluding the tag and length bytes, in decimal, is "`43, 14, 3, 2, 12`". Also see `KI_DSAPublic` and `KI_DSAPublicX957BER`.

Type of information this allows you to use:

the encoding of a DSA public key that is encoded as an X.509 SubjectPublicKeyInfo type as defined in X9.30 Part III. Note that this encoding contains all of the information specified by `KI_DSAPublic`.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BER encoding. The encoding is converted to DER before it is copied to the key object. `B_SetKeyInfo` returns `BE_WRONG_KEY_INFO` if the PublicKeyInfo specifies a public key for an algorithm other than DSA.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the DER encoding.

Can get this info type if key object already has:

`KI_DSAPublic`, `KI_DSAPublicBER`, `KI_DSAPublicX957BER`.

KI_DSAPublicX957BER

Purpose:

This KI is similar to `KI_DSAPublic` except that it uses the ASN.1 BER format. This KI allows you to parse and create an ASN.1 key info type encoded as specified by X9.57. You call `B_SetKeyInfo` to initialize a key object from the encoded key info type that includes the public component, prime, subprime, and base. You call `B_GetKeyInfo` with this KI to create an encoded key info type from a key object that was created using `KI_DSAPublic`, `KI_DSAPublicBER`, or `KI_DSAPublicX957BER`. The `OID` for DSA keys, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 206, 56, 4, 1`". Also see `KI_DSAPublic` and `KI_DSAPublicBER`.

Type of information this allows you to use:

the encoding of a DSA public key that is encoded as X9.57 SubjectPublicKeyInfo type. Note that this encoding contains all of the information specified by `KI_DSAPublic`.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BER encoding. The encoding is converted to DER before it is copied to the key object. `B_SetKeyInfo` returns `BE_WRONG_KEY_INFO` if the PublicKeyInfo specifies a public key for an algorithm other than DSA (as defined by X9.57).

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the DER encoding.

Can get this info type if key object already has:

`KI_DSAPublic`, `KI_DSAPublicBER`, `KI_DSAPublicX957BER`.

KI_ECPrivate

Purpose:

This KI allows you to specify a private key used by the Elliptic Curve algorithm. The information consists of the private component and the underlying elliptic curve parameters.

Type of information this allows you to use:

an elliptic curve private key. The parameters of the key are specified as the private component *privateKey*, and the underlying elliptic curve parameters.

Format of *info* supplied to B_SetKeyInfo:

pointer to an A_EC_PRIVATE_KEY structure:

```
typedef struct {
  A_EC_PARAMS curveParams;     /* the underlying elliptic curve parameters */
  ITEM                privateKey;                   /* private component */
} A_EC_PRIVATE_KEY;
```

Each ITEM supplies an integer in canonical format, where the ITEM's *data* points to an unsigned byte array, most significant byte first, and the ITEM's *len* gives its length. For all ITEM values except the curve parameter base, leading zeros are stripped before it is copied to the key object.

Format of *info* returned by B_GetKeyInfo:

pointer to an A_EC_PRIVATE_KEY structure.

Can get this info type if key object already has:

KI_ECPrivate or KI_ECPrivateBER.

KI_ECPrivateBER

Purpose:

This KI allows you to specify a private key used by the Elliptic Curve algorithm. The information consists of the private component and the underlying elliptic curve parameters.

Type of information this allows you to use:

the encoding of an ANSI X9.62-*style* key object identifier that defines an elliptic curve private key. ANSI X9.62 Version 1 does not define the BER encoding for a private key. Although `KI_ECPrivateBER` is X.509 compliant, it is included only as a means for the convenient internal storage of key data and is not intended for distribution in public networks.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that contains the ANSI X9.62-*style* BER encoding of a private key and its elliptic curve parameters. In compliance with X9.62, you can specify the `CHOICE` of either a full EC CURVE definition or a NAMED CURVE definition. Both ANSI X9.62 uncompressed and hybrid base points are decoded.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that contains the ANSI X9.62-*style* BER encoding of an elliptic curve's parameters. Note that the re-encoded BER will preserve the original NAMED CURVE representation of the elliptic curve parameters.

Can get this info type if key object already has:

`KI_ECPrivate` or `KI_ECPrivateBER`.

KI_ECPrivateComponent

Purpose:

This KI allows you to specify the private component of an EC private key. Unlike `KI_ECPrivate`, it does not contain the underlying EC parameters and it is not to be used with any algorithms. It provides a way to extract the EC private key.

Type of information this allows you to use:

an elliptic curve private key without its curve parameters.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that contains the private key component of an EC private key.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that contains the address and length of the private key component of an EC private key.

Restrictions:

Key objects built with this KI are not compatible with any Crypto-C AI. This KI is supplied only as a convenience to extract the EC private key component.

Can get this info type if key object already has:

`KI_ECPrivateComponent`, `KI_ECPrivateComponentBER`, `KI_ECPrivate`, or `KI_ECPrivateBER`.

KI_ECPrivateComponentBER

Purpose:

This KI allows you to specify the private component of an EC private key. Unlike `KI_ECPrivate`, it does not contain the underlying EC parameters and it is not to be used with any algorithms. It provides a way to extract the EC private key.

Type of information this allows you to use:

an elliptic curve private key without its curve parameters.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that contains the private key component of an EC private key.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that contains the address and length of an ANSI X9.62-*style* BER encoding of a private key component.

Restrictions:

Key objects built with this KI are not compatible with any Crypto-C AI. This KI is supplied only as a convenience to extract the EC private key component.

Can get this info type if key object already has:

`KI_ECPrivateComponent`, `KI_ECPrivateComponentBER`, `KI_ECPrivate`, or `KI_ECPrivateBER`.

KI_ECPublic

Purpose:

This KI allows you to specify a public key used by the Elliptic Curve algorithm. The information consists of the public component and the underlying elliptic curve parameters.

Type of information this allows you to use:

an elliptic curve public key. The parameters of the key are specified as the public component (*publicKey*), and the underlying elliptic curve parameters.

Format of *info* supplied to B_SetKeyInfo:

pointer to an A_EC_PUBLIC_KEY structure:

```
typedef struct {
  A_EC_PARAMS curveParams;     /* the underlying elliptic curve parameters */
  ITEM                 publicKey;           /* public component */
} A_EC_PUBLIC_KEY;
```

Each ITEM supplies an integer in canonical format, where the ITEM's data points to an unsigned byte array, most significant byte first, and the ITEM's len gives its length. For all ITEM values except the public component (*x*) and the curve parameter base, leading zeros are stripped before it is copied to the key object.

Format of *info* returned by B_GetKeyInfo:

pointer to an A_EC_PUBLIC_KEY structure.

Can get this info type if key object already has:

KI_ECPublic or KI_ECPublicBER.

KI_ECPublicBER

Purpose:

This KI allows you to specify a public key used by the Elliptic Curve algorithm. The information consists of the public component and the underlying elliptic curve parameters.

Type of information this allows you to use:

the encoding of an ANSI X9.62-compliant key object identifier that defines an elliptic curve public key. The parameters of the key are specified as the public component (*publicKey*), and the underlying elliptic curve parameters.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that contains the ANSI X9.62 compliant BER encoding of a public key and its elliptic curve parameters. In compliance with X9.62, you can specify the `CHOICE` of either a full EC CURVE definition or a NAMED CURVE definition. Both ANSI X9.62 uncompressed and hybrid base points are decoded.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that contains the ANSI X9.62 compliant BER encoding of an elliptic curve's parameters. Note that the re-encoded BER will preserve the original X9.62 NAMED CURVE representation of the elliptic curve parameters.

Can get this info type if key object already has:

`KI_ECPublic` or `KI_ECPublicBER`.

KI_ECPublicComponent

Purpose:

This KI allows you to specify the public key component of an EC public key. Unlike `KI_ECPublic`, it does not specify the underlying EC parameters and it is not to be used with any algorithms. It provides a way to extract the EC public key.

Type of information this allows you to use:

an elliptic curve public key without its curve parameters.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that contains the public key component of an EC public key.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that contains the address and length of the public key component of an EC public key.

Restrictions:

Key objects built with this KI are not compatible with any Crypto-C AI. This KI is supplied only as a convenience to extract the EC public component.

Can get this info type if key object already has:

`KI_ECPublicComponent`, `KI_ECPublicComponentBER`,
`KI_ECPublic`, or `KI_ECPublicBER`.

KI_ECPublicComponentBER

Purpose:

This KI allows you to specify the public key component of an EC public key. Unlike `KI_ECPublic`, it does not specify the underlying EC parameters and it is not to be used with any algorithms. It provides a way to extract the EC public key.

Type of information this allows you to use:

an elliptic curve public key without its curve parameters.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that contains the public key component of an EC public key.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that contains the address and length of an ANSI X9.62 compliant BER encoding of the public key component of an EC public key.

Restrictions:

Key objects built with this KI are not compatible with any Crypto-C AI. This KI is supplied only as a convenience to extract the EC public component.

Can get this info type if key object already has:

`KI_ECPublicComponent`, `KI_ECPublicComponentBER`,
`KI_ECPublic`, or `KI_ECPublicBER`.

KI_ExtendedToken

Purpose:

This KI allows you to specify a software-based token form of a symmetric key. See `KI_KeypairToken` for a token form of a public/private key pair.

Type of information this allows you to use:

software-based token forms of symmetric keys. Backward-compatible with `KI_Token`.

Format of *info* supplied to B_SetKeyInfo:

pointer to a `KI_EXTENDED_TOKEN_INFO` structure:

```
typedef struct {
  KI_TOKEN_INFO      keyDataStruct;
  A_X509_ATTRIB_INFO attributes;
} KI_EXTENDED_TOKEN_INFO;
```

where `A_X509_ATTRIB_INFO` is defined by:

```
typedef struct {
  A_SYMMETRIC_KEY_DEFINER externalSpecs;
  unsigned char          *keyOID;               /* Currently unimplemented */
  unsigned int            keyOIDLen;            /* Currently unimplemented */
  unsigned long           dateOfBirth;        /* When the key was created. */
               /*This time stamp currently defaults to time () function  */
} A_X509_ATTRIB_INFO;
```

and `A_SYMMETRIC_KEY_DEFINER` is defined by:

```
typedef struct {
  unsigned int keyUsage;
  unsigned int keyLengthInBytes;
  UINT4        lifeTime;
  unsigned int protectFlag;
} A_SYMMETRIC_KEY_DEFINER;
```

Format of *info* returned by B_GetKeyInfo:

pointer to a `KI_EXTENDED_TOKEN_INFO` structure (see above).

Can get this info type if key object already has:

a symmetric key of the appropriate type, for example, a DES key when using DES. Used when one of the algorithm methods used by `AI_SymKeyTokenGen` has been listed in the *`fixedChooser`* argument to `B_CreateSessionChooser`, but no hardware is present.

Notes:

`KI_ExtendedToken` can only be used if you have called `B_CreateSessionChooser` for your application.

KI_Item

Purpose:

This KI allows you to specify a generic keying material of any length. It may be used to hold a secret key of a symmetric encryption algorithm, a key of a keyed hash algorithm, a password object, etc.

Type of information this allows you to use:

a variable-length block of data (such as an RC4 key), a password for password-based encryption algorithms, or the value of a secret key when it is recovered from a public-key encryption block.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure:

```
typedef struct {
  unsigned char *data;
  unsigned int   len;
} ITEM;
```

where `data` is the address of the unsigned byte array and `len` is its length.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure (see above)

Can get this info type if key object already has:

`KI_Item`, `KI_8Byte`, `KI_24Byte`, `KI_DES8`, `KI_DES8Strong`, or `KI_DESX`.

KI_KeypairToken

Purpose:

This KI allows you to specify the software-based token form of a public and private key pair of type RSA or DSA. See `KI_ExtendedToken` for the token form of a symmetric key.

Type of information this allows you to use:

software-based token forms of RSA or DSA public and private key pairs. Backward-compatible with `KI_Token`.

Format of *info* supplied to B_SetKeyInfo:

pointer to a `KI_KEYPAIR_TOKEN_INFO` structure:

```
typedef struct {
  KI_TOKEN_INFO              keyDataStruct;
  A_X509_KEYPAIR_ATTRIB_INFO attributes;
} KI_KEYPAIR_TOKEN_INFO;
```

where `A_X509_KEYPAIR_ATTRIB_INFO` is defined by:

```
typedef struct {
  A_KEYPAIR_DEFINER externalSpecs;
  unsigned long     dateOfBirth;
} A_X509_KEYPAIR_ATTRIB_INFO;
```

and `A_KEYPAIR_DEFINER` is defined by:

```
typedef struct {
  unsigned int keyUsage;                         /* X509 key usage bit map */
  UINT4        lifeTime;              /* Key lifetime; under consideration */
  unsigned int protectFlag;                 /* Store key in encrypted form */
} A_KEYPAIR_DEFINER;
```

Format of *info* returned by B_GetKeyInfo:

pointer to a `KI_KEYPAIR_TOKEN_INFO` structure.

Can get this info type if key object already has:

an RSA or DSA key pair. Used when one of the algorithm methods used by `KI_KeypairTokenGen` has been listed in the *`fixedChooser`* argument to `B_CreateSessionChooser`, but no hardware is present.

Notes:

`KI_KeypairToken` can only be used if you have called `B_CreateSessionChooser` for your application.

KI_PKCS_RSAPrivate

Purpose:

This KI allows you to specify a private key of the RSA algorithm as defined in PKCS #1. The information consists of the modulus, exponents, two primes, and the Chinese Remainder Theorem (CRT) information. See `KI_PKCS_RSAPrivateBER` for the same key info type with BER encoding.

Type of information this allows you to use:

an RSA private key where all the integers are specified as in PKCS #1: modulus, public and private exponents, and Chinese Remainder Theorem information. Note that `KI_RSA_CRT` can be used for a private key that has the modulus and Chinese Remainder Theorem information but no public or private exponent.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `A_PKCS_RSA_PRIVATE_KEY` structure:

```
typedef struct {
  ITEM modulus;                                              /* modulus */
  ITEM publicExponent;                          /* exponent for public key */
  ITEM privateExponent;                        /* exponent for private key */
  ITEM prime[2];                                         /* prime factors */
  ITEM primeExponent[2];                   /* exponents for prime factors */
  ITEM coefficient;                                    /* CRT coefficient */
} A_PKCS_RSA_PRIVATE_KEY;
```

Each `ITEM` supplies an integer in canonical format, where the `ITEM`’s *`data`* points to an unsigned byte array, most significant byte first, and the `ITEM`’s *`len`* gives its length. All leading zeros are stripped from each integer before it is copied to the key object.

Format of *info* returned by B_GetKeyInfo:

pointer to an `A_PKCS_RSA_PRIVATE_KEY` structure (see above). All leading zeros have been stripped from each integer in the structure.

Can get this info type if key object already has:

KI_PKCS_RSAPrivate, KI_PKCS_RSAPrivateBER, or KI_RSA_CRT.

KI_PKCS_RSAPrivateBER

Purpose:

This KI is similar to `KI_PKCS_RSAPrivate` except that it uses the ASN.1 BER format. This KI allows you to parse and create an ASN.1 key info type that is encoded with the PKCS #8 standard. You call `B_SetKeyInfo` to initialize a key object from the encoded key info type that includes the modulus, exponents, two primes, and Chinese Remainder Theorem information. You call `B_GetKeyInfo` with this KI to create an encoded key info type from a key object that was created using `KI_PKCS_RSAPrivate`, `KI_PKCS_RSAPrivateBER` or `KI_RSA_CRT`. The `OID` for RSA PKCS #1 encryption, excluding the tag and length bytes, in decimal, is "`42, 134, 72, 134, 247, 13, 1, 1, 1`". Also see `KI_PKCS_RSAPrivate`.

Type of information this allows you to use:

the encoding of an RSA private key that is encoded as a PKCS #8 PrivateKeyInfo type that contains a PKCS #1 RSAPrivateKey type. Note that this encoding contains all of the information specified by `KI_PKCS_RSAPrivate`.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BER encoding. The encoding is converted to DER before it is copied to the key object. `B_SetKeyInfo` returns `BE_WRONG_KEY_INFO` if the *`PrivateKeyInfo`* specifies a private key for an algorithm other than the RSA algorithm.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the DER encoding.

Can get this info type if key object already has:

`KI_PKCS_RSAPrivate`, `KI_PKCS_RSAPrivateBER` or `KI_RSA_CRT`.

KI_RC2_BSAFE1

Purpose:

Deprecated. This type is included only for backward compatibility.

Type of information this allows you to use:

the BSAFE 1.x encoding of an RC2 key, which is the same as an SX1 key.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BSAFE 1.x encoding. `B_SetKeyInfo` returns `BE_WRONG_KEY_INFO` if the encoding specifies a secret key for an algorithm other than the RC2 algorithm (which is the same as SX1).

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BSAFE 1.x encoding.

Can get this info type if key object already has:

`KI_RC2_BSAFE1` or `KI_RC2WithBSAFE1Params`.

KI_RC2WithBSAFE1Params

Purpose:

Deprecated. This type is included only for backward compatibility.

Type of information this allows you to use:

an RC2 key (which is the same as an SX1 key) where the key value and effective key size are specified. Usually, the effective key size is a parameter to the RC2 encryption algorithm; however, BSAFE 1.x also encodes the effective key size with the RC2 key.

Format of *info* supplied to B_SetKeyInfo:

pointer to a `B_RC2_BSAFE1_PARAMS_KEY` structure:

```
typedef struct {
  unsigned char *key;                            /* pointer to 8-byte key */
  unsigned int   effectiveKeyBits;        /* effective key size parameter */
} B_RC2_BSAFE1_PARAMS_KEY;
```

`effectiveKeyBits` must be between 2 and 64, inclusive. The value of `key` is adjusted when it is copied to the key object by zeroizing unneeded bytes because `effectiveKeyBits` is smaller than 64. For example, if `effectiveKeyBits` is 32 and the hexadecimal value of `key` is 0102030405060708, then 0102030400000000 is copied to the key object because only 32 bits (4 bytes) are needed.

Format of *info* returned by B_GetKeyInfo:

pointer to a `B_RC2_BSAFE1_PARAMS_KEY` structure (see above). The value of `key` is adjusted by zeroizing unneeded bytes based on the `effectiveKeyBits` as explained above.

Can get this info type if key object already has:

`KI_RC2WithBSAFE1Params` or `KI_RC2_BSAFE1`.

KI_RSA_CRT

Purpose:

This KI allows you to specify a private key of the RSA algorithm without the public or private key exponent information. However, you must provide the modulus and Chinese Remainder Theorem information.

Type of information this allows you to use:

an RSA private key where the modulus and Chinese Remainder Theorem information integers are specified, but not the public or private exponent integers. (For RSA private key information with the public and private exponent integers, see KI_PKCS_RSAPrivate.)

Format of *info* supplied to B_SetKeyInfo:

pointer to an A_RSA_CRT_KEY structure:

```
typedef struct {
  ITEM modulus;                                              /* modulus */
  ITEM prime[2];                                       /* prime factors */
  ITEM primeExponent[2];                 /* exponents for prime factors */
  ITEM coefficient;                                   /* CRT coefficient */
} A_RSA_CRT_KEY;
```

Each ITEM supplies an integer in canonical format, where the ITEM's *data* points to an unsigned byte array, most significant byte first, and the ITEM's *len* gives its length. All leading zeros are stripped from each integer before it is copied to the key object.

Format of *info* returned by B_GetKeyInfo:

pointer to an A_RSA_CRT_KEY structure (see above). All leading zeros have been stripped from each integer in the structure.

Can get this info type if key object already has:

KI_RSA_CRT, KI_PKCS_RSAPrivate, or KI_PKCS_RSAPrivateBER.

KI_RSAPrivate

Purpose:

This KI allows you to specify an RSA private key with the modulus and private key exponent. Unlike `KI_PKCS_RSAPrivate`, it does not contain the Chinese Remainder Theorem information and it is not used with any algorithms. It provides a way to store or transport a private key.

Type of information this allows you to use:

an RSA private key where the modulus and private exponent integers are specified, but the Chinese Remainder Theorem information is not.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `A_RSA_KEY` structure:

```
typedef struct {
  ITEM modulus;                                              /* modulus */
  ITEM exponent;                                            /* exponent */
} A_RSA_KEY;
```

Each `ITEM` supplies an integer in canonical format, where the `ITEM`'s *`data`* points to an unsigned byte array, most significant byte first, and the `ITEM`'s *`len`* gives its length. All leading zeros are stripped from each integer before it is copied to the key object.

Format of *info* returned by B_GetKeyInfo:

pointer to an `A_RSA_KEY` structure (see above). All leading zeros have been stripped from each integer in the structure.

Can get this info type if key object already has:

`KI_RSAPrivate`, `KI_PKCS_RSAPrivate` or `KI_PKCS_RSAPrivateBER`.

Note:

You can use `KI_RSAPrivate` to set a key object with your private modulus and private exponent. This can be used for storing your key information or when you need to export the information in “raw” form to another application. However, there are no Crypto-C algorithms that use this key type.

KI_RSAPrivateBSAFE1

Purpose:

Deprecated. This type is included only for backward compatibility.

Type of information this allows you to use:

the BSAFE 1.x encoding of an RSA private key.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BSAFE 1.x encoding. `B_SetKeyInfo` returns `BE_WRONG_KEY_INFO` if the encoding specifies a private key for an algorithm other than RSA.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BSAFE 1.x encoding.

Can get this info type if key object already has:

`KI_RSAPrivateBSAFE1`, `KI_RSA_CRT`, `KI_PKCS_RSAPrivate` or `KI_PKCS_RSAPrivateBER`.

KI_RSAPublic

Purpose:

This KI allows you to specify an RSA public key with the modulus and public key exponent. See KI_RSAPublicBER for the same key info type with BER encoding.

Type of information this allows you to use:

an RSA public key where the modulus and exponent integers are specified.

Format of *info* supplied to B_SetKeyInfo:

pointer to an A_RSA_KEY structure:

```
typedef struct {
  ITEM modulus;                                                  /* modulus */
  ITEM exponent;                                                /* exponent */
} A_RSA_KEY;
```

Each ITEM supplies an integer in canonical format, where the ITEM's *data* points to an unsigned byte array, most significant byte first, and the ITEM's *len* gives its length. All leading zeros are stripped from each integer before it is copied to the key object.

Format of *info* returned by B_GetKeyInfo:

pointer to an A_RSA_KEY structure (see above). All leading zeros have been stripped from each integer in the structure.

Can get this info type if key object already has:

KI_RSAPublic, KI_RSAPublicBER, KI_PKCS_RSAPrivate, or KI_PKCS_RSAPrivateBER.

KI_RSAPublicBER

Purpose:

This KI is similar to `KI_RSAPublic` except that it uses the ASN.1 BER format. This KI allows you to parse and create an ASN.1 key info type that is encoded with the X.509 standard of SubjectPublicKeyInfo. You call `B_SetKeyInfo` to initialize a key object from the encoded key info type that includes the modulus and public exponent. You call `B_GetKeyInfo` with this KI to create an encoded key info type from a key object that was created using `KI_RSAPublic`, `KI_RSAPublicBER`, `KI_PKCS_RSAPrivate` or `KI_PKCS_RSAPrivateBER`. The `OID` for RSA PKCS #1 encryption, excluding the tag and length bytes, in decimal, is “`42, 134, 72, 134, 247, 13, 1, 1, 1`”. Also see `KI_RSAPublic`.

Type of information this allows you to use:

the encoding of an RSA public key that is encoded as an X.509 SubjectPublicKeyInfo type that contains an X.509 RSAPublicKey type (also defined in PKCS #1). Note that this encoding contains all of the information specified by `KI_RSAPublic`.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BER encoding. The encoding is converted to DER before it is copied to the key object. `B_SetKeyInfo` returns `BE_WRONG_KEY_INFO` if the public key info specifies a public key for an algorithm other than the RSA algorithm. Note that `B_SetKeyInfo` will accept an encoding that contains an object identifier for `rsa` as well as `rsaEncryption` (defined in PKCS #1).

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the DER encoding. Note that `B_GetKeyInfo` returns an encoding that contains the object identifier for `rsaEncryption` (defined in PKCS #1) as opposed to `rsa`.

Can get this info type if key object already has:

`KI_RSAPublicBER`, `KI_RSAPublic`, `KI_PKCS_RSA`Private, or `KI_PKCS_RSAPrivateBER`.

KI_RSAPublicBSAFE1

Purpose:

Deprecated. This type is included only for backward compatibility.

Type of information this allows you to use:

the BSAFE 1.x encoding of an RSA public key.

Format of *info* supplied to B_SetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BSAFE 1.x encoding. `B_SetKeyInfo` returns `BE_WRONG_KEY_INFO` if the encoding specifies a public key for an algorithm other than the RSA algorithm.

Format of *info* returned by B_GetKeyInfo:

pointer to an `ITEM` structure that gives the address and length of the BSAFE 1.x encoding.

Can get this info type if key object already has:

`KI_RSAPublicBSAFE1`, `KI_RSAPublic`, `KI_PKCS_RSAPrivate`, `KI_PKCS_RSAPrivateBER` or `KI_RSAPublicBER`.

KI_Token

Purpose:

This KI allows you to specify a hardware-based token form of a key, which may be either a symmetric key or a public/private key pair. Also see `KI_ExtendedToken` and `KI_KeypairToken` for other key info types with token forms.

Type of information this allows you to use:

hardware-based token forms of symmetric keys and public/private key pairs.

Format of *info* supplied to B_SetKeyInfo:

pointer to a `KI_TOKEN_INFO` structure:

```
typedef struct {
  ITEM manufacturerId;                       /* tag used to differentiate */
                                             /* different hardware tokens */
  ITEM internalKey;                            /* OEM-supplied key handle */
} KI_TOKEN_INFO;
```

Format of *info* returned by B_GetKeyInfo:

pointer to a `KI_TOKEN_INFO` structure (see above).

Can get this info type if key object already has:

a key object of the appropriate type, for example, an RSA key pair for RSA operations or a DES key for DES operations. Hardware that uses key tokens must be present.

Notes:

Can only be used in conjunction with a hardware implementation; in particular, `KI_Token` can only be used if you have called `B_CreateSessionChooser` for your application.

Chapter 4

Details of Crypto-C Functions

This section describes the toolkit's top level API and its bottom level platform-specific routines. The procedures with names that start “`B_`” make up the top level API which is called by applications. The procedure names starting with “`T_`” are called by the toolkit's internal routines to perform platform-specific operations like allocating and copying memory.

B_BuildTableFinal

```
int B_BuildTableFinal (
  B_ALGORITHM_OBJ   buildTableObj,                /* table-building object */
  unsigned char    *accelTable,               /* acceleration table buffer */
  unsigned int     *accTableByteLen,                             /* size of */
                                   /*  created acceleration table in bytes */
  unsigned int      maxAccTableLength,                           /* size of */
                                              /* acceleration table buffer */
  A_SURRENDER_CTX * surrenderCtx                       /* surrender context */
);
```

Description

Generates and outputs the acceleration table to *accelTable,* setting *accTableByteLen* to the number of bytes output. It returns BE_OUTPUT_LEN if *maxAccTableLength* is smaller than needed.

Return value

Value	Description
0	Operation was successful.
non-zero	Error. See Appendix A, "Crypto-C Error Types."

B_BuildTableGetBufSize

```
int B_BuildTableGetBufSize (
  B_ALGORITHM_OBJ buildTableObj,                 /* table-building object */
  unsigned int   *tableSizeInBytes              /* size of table in bytes */
);
```

Description

Sets `tableSizeInBytes` to the buffer size needed to accommodate the generated table.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, “Crypto-C Error Types”

B_BuildTableInit

```
int B_BuildTableInit (
  B_ALGORITHM_OBJ     buildTableObj,            /* table-building object */
  B_ALGORITHM_CHOOSER algorithmChooser,             /* algorithm chooser */
  A_SURRENDER_CTX     surrenderCtx                  /* surrender context */
);
```

Description

Initializes a table-building object used to build acceleration tables for elliptic curve cryptography.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, “Crypto-C Error Types”

B_CreateAlgorithmObject

```
int B_CreateAlgorithmObject (
  B_ALGORITHM_OBJ *algorithmObject                 /* new algorithm object */
);
```

Description

`B_CreateAlgorithmObject` allocates and initializes a new algorithm object, storing the result in `algorithmObject`. If `B_CreateAlgorithmObject` is unsuccessful, no memory is allocated and `algorithmObject` is set to `(B_ALGORITHM_OBJ)NULL_PTR`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, “Crypto-C Error Types”

B_CreateKeyObject

```
int B_CreateKeyObject (
  B_KEY_OBJ *keyObject                                  /* new key object */
);
```

Description

`B_CreateKeyObject` allocates and initializes a new key object, storing the result in `keyObject`. If `B_CreateKeyObject` is unsuccessful, no memory is allocated and `keyObject` is set to (`B_KEY_OBJ`)`NULL_PTR`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, “Crypto-C Error Types”

B_CreateSessionChooser

```
int B_CreateSessionChooser (
B_ALGORITHM_CHOOSER  fixedChooser,                /* Chooser consisting of */
                                         software-based  algorithm methods. */
B_ALGORITHM_CHOOSER *sessionChooser,                     /* Runtime chooser */
                 /* dynamically bound to available hardware based methods. */

HW_TABLE_ENTRY      *staticHardwareList[ ],  /* List of statically defined */
                                       /* hardware methods terminated by a */
                                               /*  properly cast NULL_PTR. */
ITEM                *passPhrase,                    /* hardware passphrase */
POINTER             *amTagList,                /* For now pass (*)NULL_PTR */
unsigned char     ***listOfOEMTags             /* Returns list of OEM tags */
                                          /* for methods in sessionChooser */
);
```

Description

`B_CreateSessionChooser` replicates the fixed chooser inclusive of making private copies of the AM structures. Whenever possible the software-based methods are replaced with the hardware-based methods defined either statically by *`staticHardwareList`* or dynamically determined by platform specific routines. All methods in the *`fixedChooser`* will be represented in the *`sessionChooser`* and will appear multiple times if there are multiple hardware substitutes available.

Notes:

It is simple to limit the binding of hardware methods to exclusively static or dynamically defined entries. The algorithm method structures that constitute *`sessionChooser`* are `B_METHOD` structures extended to include an information pointer and a finalization function to destroy it.

Return value

Value	Description
0	Operation was successful.
nonzero	unsuccessful, allocation error

B_DecodeDigestInfo

```
int B_DecodeDigestInfo  (
  ITEM          *algorithmID,       /* message digest algorithm identifier */
  ITEM          *digest,                                /* message digest value */
  unsigned char *digestInfo,                             /* digestInfo encoding */
  unsigned int   digestInfoLen                          /* length of digestInfo */
);
```

Description

`B_DecodeDigestInfo` decodes the BER encoding of a PKCS #1 DigestInfo type that is given by *`digestInfo`* of length *`digestInfoLen`*. On output, the *`algorithmID`* `ITEM` gives the digest algorithm identifier and the *`digest`* `ITEM` gives the digest value. The `ITEM` type is defined by the `KI_Item` key info type on page 273.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_DecodeFinal

```
int B_DecodeFinal (
  B_ALGORITHM_OBJ algorithmObject,                    /* algorithm object */
  unsigned char  *partOut,                          /* output data buffer */
  unsigned int   *partOutLen,                    /* length of output data */
  unsigned int    maxPartOutLen              /* size of output data buffer */
);
```

Description

`B_DecodeFinal` finalizes the decoding process specified by *`algorithmObject`*, writing any remaining decoded output to *`partOut`*, which is a buffer supplied by the caller of at least *`maxPartOutLen`* bytes, and setting *`partOutLen`* to the number of bytes written to *`partOut`*. *`algorithmObject`* is reset to the state it was in after the call to `B_DecodeInit`, so that another decoding process may be performed. See `B_DecodeInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_DecodeInit

```
int B_DecodeInit (
  B_ALGORITHM_OBJ algorithmObject                    /* algorithm object */
);
```

Description

`B_DecodeInit` allocates and initializes `algorithmObject` for decoding (not decrypting) data using the algorithm specified by a previous call to `B_SetAlgorithmInfo`. For example, the `AI_RFC1113Recode` algorithm provides Base64 encoding and decoding to convert binary data to and from a printable form suitable for most email systems. Notice that there are no cryptographic keys for encoding or decoding.

`B_DecodeInit` only needs to be called once to set up a decode algorithm. The `B_DecodeUpdate` routine can be called multiple times to process blocks of data, and `B_DecodeFinal` is called once to process the last block which includes removing any trailing pad bytes. After `B_DecodeFinal` is called, `B_DecodeUpdate` can be called to start decoding another sequence of blocks. There is no need to call `B_DecodeInit` again.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, “Crypto-C Error Types”

B_DecodeUpdate

```
int B_DecodeUpdate  (
  B_ALGORITHM_OBJ algorithmObject,                     /* algorithm object */
  unsigned char  *partOut,                          /* output data buffer */
  unsigned int   *partOutLen,                    /* length of output data */
  unsigned int    maxPartOutLen,          /* size of output data buffer */
  unsigned char  *partIn,                                 /* input data */
  unsigned int    partInLen                      /* length of output data */
);
```

Description

`B_DecodeUpdate` updates the decoding process specified by *`algorithmObject`* with *`partInLen`* bytes from *`partIn`*, writing the decoded output to *`partOut`*, which is a buffer supplied by the caller of at least *`maxPartOutLen`* bytes, and setting *`partOutLen`* to the number of bytes written to *`partOut`*. `B_DecodeUpdate` may be called zero or more times to supply the data by parts. See `B_DecodeInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, “Crypto-C Error Types”

B_DecryptFinal

```
int B_DecryptFinal (
  B_ALGORITHM_OBJ  algorithmObject,                    /* algorithm object */
  unsigned char   *partOut,                          /* output data buffer */
  unsigned int    *partOutLen,                     /* length of output data */
  unsigned int     maxPartOutLen,             /* size of output data buffer */
  B_ALGORITHM_OBJ  randomAlgorithm,                  /* random byte source */
  A_SURRENDER_CTX *surrenderContext                   /* surrender context */
);
```

Description

`B_DecryptFinal` finalizes the decrypting process specified by *`algorithmObject`*, writing any remaining decrypted output to *`partOut`*, which is a buffer supplied by the caller of at least *`maxPartOutLen`* bytes, and setting *`partOutLen`* to the number of bytes written to *`partOut`*. The algorithm object for supplying random numbers is *`randomAlgorithm`*; it may be `(B_ALGORITHM_OBJ)NULL_PTR` for decrypting algorithms that do not need random numbers. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. *`algorithmObject`* is reset to the state it was in after the call to `B_DecryptInit`, so that another decrypting process may be performed. See `B_DecryptInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_DecryptInit

```
int B_DecryptInit (
  B_ALGORITHM_OBJ      algorithmObject,                  /* algorithm object */
  B_KEY_OBJ            keyObject,                              /* key object */
  B_ALGORITHM_CHOOSER algorithmChooser,                 /* algorithm chooser */
  A_SURRENDER_CTX    *surrenderContext                  /* surrender context */
);
```

Description

`B_DecryptInit` initializes `algorithmObject` for decrypting data using the algorithm specified by a previous call to `B_SetAlgorithmInfo`. The key object for supplying the key information is `keyObject`. The chooser for selecting the algorithm method is *`algorithmChooser`*. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it.

`B_DecryptInit` only needs to be called once to set up a key. The `B_DecryptUpdate` routine can be called multiple times to process blocks of data, and `B_DecryptFinal` is called once to process the last block, which includes removing the trailing pad bytes. After `B_DecryptFinal` is called, `B_DecryptUpdate` can be called to start processing another sequence of blocks that were encrypted in the same key. If a different CBC Initialization Vector (IVs) is used with each sequence of blocks, `B_SetAlgorithmInfo` can be called with `AI_CBC_IV8` to set the new IV before calling `B_DecryptUpdate`. There is no need to call `B_DecryptInit` again with the same key.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_DecryptUpdate

```
int B_DecryptUpdate (
  B_ALGORITHM_OBJ  algorithmObject,                  /* algorithm object */
  unsigned char   *partOut,                        /* output data buffer */
  unsigned int    *partOutLen,                   /* length of output data */
  unsigned int     maxPartOutLen,            /* size of output data buffer */
  unsigned char   *partIn,                                 /* input data */
  unsigned int     partInLen,                      /* length of input data */
  B_ALGORITHM_OBJ  randomAlgorithm,                  /* random byte source */
  A_SURRENDER_CTX *surrenderContext                    /* surrender context */
);
```

Description

`B_DecryptUpdate` updates the decrypting process specified by *`algorithmObject`* with *`partInLen`* bytes from *`partIn`*, writing the decrypted output to *`partOut`*, which is a buffer supplied by the caller of at least *`maxPartOutLen`* bytes, and setting *`partOutLen`* to the number of bytes written to *`partOut`*. The algorithm object for supplying random numbers is *`randomAlgorithm`*; it may be `(B_ALGORITHM_OBJ)NULL_PTR` for decrypting algorithms that do not need random numbers. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. `B_DecryptUpdate` may be called zero or more times to supply the data by parts. See `B_DecryptInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_DestroyAlgorithmObject

```
void B_DestroyAlgorithmObject (
  B_ALGORITHM_OBJ *algorithmObject          /* pointer to algorithm object */
);
```

Description

B_DestroyAlgorithmObject destroys *algorithmObject*, zeroizing any sensitive information, freeing the memory the algorithm object occupied, and setting *algorithmObject* to (B_ALGORITHM_OBJ)NULL_PTR. If *algorithmObject* is already (B_ALGORITHM_OBJ)NULL_PTR or is not a valid algorithm object, no action is taken. See B_CreateAlgorithmObject.

After this routine is called, all the pointers to information blocks returned by calls to B_GetAlgorithmInfo will no longer be valid, since the memory associated with those blocks will have been zeroed and freed.

Return value

There is no return value.

B_DestroyKeyObject

```
void B_DestroyKeyObject (
  B_KEY_OBJ *keyObject                             /* pointer to key object */
);
```

Description

`B_DestroyKeyObject` destroys *`keyObject`*, zeroizing any sensitive information, freeing the memory the key object occupied, and setting *`keyObject`* to `(B_KEY_OBJ)NULL_PTR`. If *`keyObject`* is already `(B_KEY_OBJ)NULL_PTR` or is not a valid key object, no action is taken. See `B_CreateKeyObject`.

After this routine is called, all the pointers to information blocks returned by calls to `B_GetKeyInfo` will no longer be valid, since the memory associated with those blocks will have been zeroed and freed.

Return value

There is no return value.

B_DigestFinal

```
int B_DigestFinal (
  B_ALGORITHM_OBJ  algorithmObject,                        /* algorithm object */
  unsigned char   *digest,                    /* message digest output buffer */
  unsigned int    *digestLen,             /* length of message digest output */
  unsigned int     maxDigestLen,                     /* size of output buffer */
  A_SURRENDER_CTX *surrenderContext                       /* surrender context */
);
```

Description

`B_DigestFinal` finalizes the digesting process for *`algorithmObject`* and writes the message digest to *`digest`*, which is a buffer supplied by the caller of at least *`maxDigestLen`* bytes, and sets *`digestLen`* to the length of the digest. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. *`algorithmObject`* is reset to the state it was in after the call to `B_DigestInit`, so that another message digesting process may be performed.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_DigestInit

```
int B_DigestInit (
  B_ALGORITHM_OBJ     algorithmObject,              /* algorithm object */
  B_KEY_OBJ           keyObject,                          /* key object */
  B_ALGORITHM_CHOOSER algorithmChooser,            /* algorithm chooser */
  A_SURRENDER_CTX    *surrenderContext             /* surrender context */
);
```

Description

`B_DigestInit` initializes *`algorithmObject`* for computing a message digest using the algorithm specified by a previous call to `B_SetAlgorithmInfo`. The chooser for selecting the algorithm method is *`algorithmChooser`*. The key object for supplying the key information is *`keyObject`*; it should be `(B_KEY_OBJ)NULL_PTR` for keyless digesting algorithms. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it.

`B_DigestInit` only needs to be called once to set up a digest algorithm. The `B_DigestUpdate` routine can be called multiple times to process blocks of data, and `B_DigestFinal` is called once to process the last block which includes producing the result. After `B_DigestFinal` is called, `B_DigestUpdate` can be called to start digesting another sequence of blocks. There is no need to call `B_DigestInit` again.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_DigestUpdate

```
int B_DigestUpdate (
  B_ALGORITHM_OBJ  algorithmObject,                    /* algorithm object */
  unsigned char   *partIn,                                    /* input data */
  unsigned int     partInLen,                       /* length of input data */
  A_SURRENDER_CTX *surrenderContext                     /* surrender context */
);
```

Description

`B_DigestUpdate` updates *`algorithmObject`* with *`partInLen`* bytes from *`partIn`*. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. `B_DigestUpdate` may be called zero or more times to supply the data by parts. See `B_DigestInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_EncodeDigestInfo

```
int B_EncodeDigestInfo (
  unsigned char *digestInfo,                        /* encoded output buffer */
  unsigned int  *digestInfoLen,                  /* length of encoded output */
  unsigned int   maxDigestInfoLen,              /* size of digestInfo buffer */
  ITEM          *algorithmID,      /* message digest algorithm identifier */
  unsigned char *digest,                              /* message digest value */
  unsigned int   digestLen                              /* length of digest */
);
```

Description

B_EncodeDigestInfo encodes the DER encoding of a PKCS #1 DigestInfo type, writing the encoding to *digestInfo*, which is a buffer supplied by the caller of at least *maxDigestInfoLen* bytes, and sets *digestInfoLen* to the length of the encoding. *algorithmID* points to an ITEM that gives the DER encoding of the message digest algorithm. The ITEM type is defined by the KI_Item key info type in Chapter 3. Typically, *algorithmID* is the value of *info* returned by calling B_GetAlgorithmInfo. *digest* points to the message digest value of length *digestLen*. Typically, *digest* is returned by B_DigestFinal.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_EncodeFinal

```
int B_EncodeFinal (
  B_ALGORITHM_OBJ algorithmObject,                    /* algorithm object */
  unsigned char  *partOut,                          /* output data buffer */
  unsigned int   *partOut Len,                   /* length of output data */
  unsigned int    maxPartOutLen              /* size of output data buffer */
);
```

Description

`B_EncodeFinal` finalizes the encoding process specified by *`algorithmObject`*, writing any remaining encoded output to *`partOut`*, which is a buffer supplied by the caller of at least *`maxPartOutLen`* bytes, and setting *`partOutLen`* to the number of bytes written to *`partOut`*. *`algorithmObject`* is reset to the state it was in after the call to `B_EncodeInit`, so that another encoding process may be performed. See `B_EncodeInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_EncodeInit

```
int B_EncodeInit (
  B_ALGORITHM_OBJ algorithmObject                     /* algorithm object */
);
```

Description

`B_EncodeInit` initializes *`algorithmObject`* for encoding data using the algorithm specified by a previous call to `B_SetAlgorithmInfo`. For example, the `AI_RFC1113Recode` algorithm provides Base64 encoding and decoding to convert binary data to and from a printable form suitable for most email systems. Notice that there are no cryptographic keys for encoding or decoding.

`B_EncodeInit` only needs to be called once to set up an Encode algorithm. The `B_EncodeUpdate` routine can be called multiple times to process blocks of data, and `B_EncodeFinal` is called once to process the last block which includes adding any trailing pad bytes. After `B_EncodeFinal` is called, `B_EncodeUpdate` can be called to start decoding another sequence of blocks. There is no need to call `B_EncodeInit` again.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_EncodeUpdate

```
int B_EncodeUpdate (
  B_ALGORITHM_OBJ algorithmObject,                    /* algorithm object */
  unsigned char  *partOut,                         /* output data buffer */
  unsigned int   *partOutLen,                   /* length of output data */
  unsigned int    maxPartOutLen,           /* size of output data buffer */
  unsigned char  *partIn,                                  /* input data */
  unsigned int    partInLen                      /* length of input data */
);
```

Description

`B_EncodeUpdate` updates the encoding process specified by *`algorithmObject`* with *`partInLen`* bytes from *`partIn`*, writing the encoded output to *`partOut`*, which is a buffer supplied by the caller of at least *`maxPartOutLen`* bytes, and setting *`partOutLen`* to the number of bytes written to *`partOut`*. `B_EncodeUpdate` may be called zero or more times to supply the data by parts. See `B_EncodeInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_EncryptFinal

```
int B_EncryptFinal (
  B_ALGORITHM_OBJ  algorithmObject,                    /* algorithm object */
  unsigned char   *partOut,                         /* output data buffer */
  unsigned int    *partOutLen,                   /* length of output data */
  unsigned int     maxPartOutLen,           /* size of output data buffer */
  B_ALGORITHM_OBJ  randomAlgorithm,                 /* random byte source */
  A_SURRENDER_CTX *surrenderContext                 /* surrender context */
);
```

Description

`B_EncryptFinal` finalizes the encrypting process specified by *`algorithmObject`*, writing any remaining encrypted output to *`partOut`*, which is a buffer supplied by the caller of at least *`maxPartOutLen`* bytes, and setting *`partOutLen`* to the number of bytes written to *`partOut`*. The algorithm object for supplying random numbers is *`randomAlgorithm`*; it may be `(B_ALGORITHM_OBJ)NULL_PTR` for encrypting algorithms that do not need random numbers. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. *`algorithmObject`* is reset to the state it was in after the call to `B_EncryptInit`, so that another encrypting process may be performed. See `B_EncryptInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_EncryptInit

```
int B_EncryptInit (
  B_ALGORITHM_OBJ     algorithmObject,                /* algorithm object */
  B_KEY_OBJ           keyObject,                            /* key object */
  B_ALGORITHM_CHOOSER algorithmChooser,              /* algorithm chooser */
  A_SURRENDER_CTX    *surrenderContext               /* surrender context */
);
```

Description

`B_EncryptInit` initializes *`algorithmObject`* for encrypting data using the algorithm specified by a previous call to `B_SetAlgorithmInfo`. The key object for supplying the key information is *`keyObject`*. The chooser for selecting the algorithm method is *`algorithmChooser`*. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it.

`B_EncryptInit` only needs to be called once to set up a key. The `B_EncryptUpdate` routine can be called multiple times to process blocks of data, and `B_EncryptFinal` is called once to process the last block, which includes adding the trailing pad bytes. After `B_EncryptFinal` is called, `B_EncryptUpdate` can be called to start processing another sequence of blocks. If a different CBC Initialization Vector (IVs) is used with each sequence of blocks, `B_SetAlgorithmInfo` can be called with `AI_CBC_IV8` to set the new IV before calling `B_EncryptUpdate`. There is no need to call `B_EncryptInit` again with the same key.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_EncryptUpdate

```
int B_EncryptUpdate (
  B_ALGORITHM_OBJ  algorithmObject,                  /* algorithm object */
  unsigned char   *partOut,                        /* output data buffer */
  unsigned int    *partOutLen,                   /* length of output data */
  unsigned int     maxPartOutLen,          /* size of output data buffer */
  unsigned char   *partIn,                                /* input data */
  unsigned int     partInLen,                      /* length of input data */
  B_ALGORITHM_OBJ  randomAlgorithm,                 /* random byte source */
  A_SURRENDER_CTX *surrenderContext                   /* surrender context */
);
```

Description

`B_EncryptUpdate` updates the encrypting process specified by *`algorithmObject`* with *`partInLen`* bytes from *`partIn`*, writing the encrypted output to *`partOut`*, which is a buffer supplied by the caller of at least *`maxPartOutLen`* bytes, and setting *`partOutLen`* to the number of bytes written to *`partOut`*. The algorithm object for supplying random numbers is *`randomAlgorithm`*; it may be `(B_ALGORITHM_OBJ)NULL_PTR` for encrypting algorithms that do not need random numbers. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. `B_EncryptUpdate` may be called zero or more times to supply the data by parts.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_FreeSessionChooser

```
int B_FreeSessionChooser (
  B_ALGORITHM_CHOOSER *sessionChooser,       /* Address of runtime chooser */
                                         /* dynamically bound to available */
                                                 /* hardware-based methods */
  unsigned char     ***oemTagList,              /* Address of list of OEM */
                                                   /* hardware method tags */
);
```

Description

`B_FreeSessionChooser` frees the memory allocated in the process of creating *`sessionChooser`* and *`oemTagList`*. Whenever a non-null information pointer from the extended AM is encountered, its corresponding Final function is called to destroy it.

Return value

Value	Description
0	Operation was successful.
nonzero	unsuccessful, allocation error

B_GenerateInit

```
int B_GenerateInit (
  B_ALGORITHM_OBJ     algorithmObject,                /* algorithm object */
  B_ALGORITHM_CHOOSER algorithmChooser,              /* algorithm chooser */
  A_SURRENDER_CTX    *surrenderContext               /* surrender context */
);
```

Description

`B_GenerateInit` initializes *`algorithmObject`* using the algorithm specified by a previous call to `B_SetAlgorithmInfo`. The chooser for selecting the algorithm method is *`algorithmChooser`*. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it.

This routine is used to initialize several of the toolkit's parameter generation algorithms like `B_GenerateKeypair` or `B_GenerateParameters`. However, `B_RandomInit` is used to initialize the generator for `B_GenerateRandomBytes`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_GenerateKeypair

```
int B_GenerateKeypair (
  B_ALGORITHM_OBJ  algorithmObject,                    /* algorithm object */
  B_KEY_OBJ        publicKey,                            /* new public key */
  B_KEY_OBJ        privateKey,                          /* new private key */
  B_ALGORITHM_OBJ  randomAlgorithm,                    /* random algorithm */
  A_SURRENDER_CTX *surrenderContext                   /* surrender context */
);
```

Description

`B_GenerateKeypair` uses `algorithmObject` to generate a keypair, setting *`publicKey`* and *`privateKey`* to the result. The algorithm object for supplying random numbers is *`randomAlgorithm`*. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. *`algorithmObject`* is reset to the state it was in after the call to `B_GenerateInit`, so that another keypair generation may be performed. See `B_GenerateInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_GenerateParameters

```
int B_GenerateParameters  (
  B_ALGORITHM_OBJ  algorithmObject,                    /* algorithm object */
  B_ALGORITHM_OBJ  resultAlgorithmObject,       /* result algorithm object */
  B_ALGORITHM_OBJ  randomAlgorithmObject,              /* random algorithm */
  A_SURRENDER_CTX *surrenderContext                   /* surrender context */
);
```

Description

`B_GenerateParameters` uses *`algorithmObject`* to generate algorithm parameters, setting *`resultAlgorithmObject`* to the result. The application may then use `B_GetAlgorithmInfo` to get the new parameters from *`resultAlgorithmObject`*. The algorithm object for supplying random numbers is *`randomAlgorithmObject`*. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. *`algorithmObject`* is reset to the state it was in after the call to `B_GenerateInit`, so that another parameter generation may be performed. See `B_GenerateInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_GenerateRandomBytes

```
int B_GenerateRandomBytes (
  B_ALGORITHM_OBJ  randomAlgorithm,                    /* random algorithm */
  unsigned char   *output,                      /* buffer for output bytes */
  unsigned int     outputLen,                 /* number of bytes to output */
  A_SURRENDER_CTX *surrenderContext                    /* surrender context */
);
```

Description

`B_GenerateRandomBytes` generates *`outputLen`* pseudo-random bytes from *`randomAlgorithm`*, storing the result in *`output`*. The *`randomAlgorithm`* must have been seeded. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. See `B_RandomInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_GetAlgorithmInfo

```
int B_GetAlgorithmInfo (
  POINTER          *info,                        /* algorithm information */
  B_ALGORITHM_OBJ algorithmObject,                    /* algorithm object */
  B_INFO_TYPE      infoType              /* type of algorithm information */
);
```

Description

`B_GetAlgorithmInfo` gets the information held by *`algorithmObject`* in the format specified by *`infoType`*, storing the result in *`info`*. The value of *`infoType`* is one of the algorithm info types with an `AI_` prefix listed in Chapter 2. The format of the information returned by `B_GetAlgorithmInfo` is the same as the format supplied to `B_SetAlgorithmInfo`.

This routine can be used to convert between external representations of information. For example, an algorithm can be set up to perform encryption using `AI_MD2WithDES_CBCPad`, and later the BER representation of that algorithm can be computed by calling `B_GetAlgorithmInfo` with `AI_MD2WithDES_CBCPadBER`. These external representations of algorithm identifiers are used in RSA Security's PKCS standards and other industry cryptographic standards.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_GetExtendedErrorInfo

```
void B_GetExtendedErrorInfo (
  B_ALGORITHM_OBJ algorithmObj,                        /* algorithm object */
  ITEM           *errorDataItem,         /* returns pointer to error data */
                                                   /* and length of data */
  POINTER        *AM                      /* Set to point at method table */
);
```

Description

`B_GetExtendedErrorInfo` sets *`errorDataItem->data`* to point at the error data, and *`errorDataItem->len`* to the length in bytes of the error data. `AM` is set to the address of the algorithm method that originated the error, if any.

Notes:

A `NULL` extended error is indicated by a length of zero. The error data may in reality be a data structure that includes pointers to allocated memory. These allocations are cleaned up using an error destruction routine assigned during the creation of the extended error data.

Return value

There is no return value.

B_GetKeyExtendedErrorInfo

```
void B_GetKeyExtendedErrorInfo (
  B_KEY_OBJ keyObject,                                      /* key object */
  ITEM     *errorDataItem,                            /* returns pointer to */
                                                /* error data and length of data */
  POINTER  *AM;                                  /* Set to point at method table */
);
```

Description

B_GetKeyExtendedErrorInfo sets *errorDataItem->data* to point at the error data, and *errorDataItem->len* to the length in bytes of the error data. AM is optionally set by the hardware manufacturer; consult the documentation supplied by your hardware vendor for more information.

Notes:

A NULL extended error is indicated by a length of zero. The error data may in reality be a data structure that includes pointers to allocated memory. These allocations are cleaned up using an error destruction routine assigned during the creation of the extended error data.

Return value

There is no return value.

B_GetKeyInfo

```
int B_GetKeyInfo (
  POINTER    *info,                                  /* key information */
  B_KEY_OBJ   keyObject,                                  /* key object */
  B_INFO_TYPE infoType                      /* type of key information */
);
```

Description

`B_GetKeyInfo` gets the information held by *`keyObject`* in the format specified by *`infoType`*, storing the result in *`info`*. The value of *`infoType`* is one of the key info types with a `KI_` prefix listed in Chapter 3. The format of the information returned by `B_GetKeyInfo` is not always identical to the format supplied to `B_SetKeyInfo` because `B_SetKeyInfo` may convert the key data to canonical form before storing it in the key object. For example, `KI_DES8` sets the DES parity before storing, `KI_RSAPublicBER` converts a BER encoding to DER, and `KI_RSAPublic` strips off leading zeros from the modulus and exponent integers.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_IntegerBits

```
unsigned int B_IntegerBits (
  unsigned char *integer,                          /* canonical integer */
  unsigned int   integerLen                            /* length in bytes */
);
```

Description

`B_IntegerBits` returns the number of significant bits in an arbitrary-length integer, where *`integer`* points to an unsigned byte array, most significant byte, first and *`integerLen`* gives its length. Leading zeroes are ignored. The integer is considered unsigned; that is, the most-significant bit is counted and is not considered a sign bit. If *`integerLen`* is zero, *`integer`* is ignored and `B_IntegerBits` returns zero. A typical application uses `B_IntegerBits` to determine the key size in bits of an RSA key by passing in the modulus.

This routine can be used to examine the value of a large integer such as the ones returned by `B_GetKeyInfo` for KI's like `KI_RSAPublic`.

Return value

`B_IntegerBits` returns the number of significant bits in *`integer`*.

B_KeyAgreeInit

```
int B_KeyAgreeInit (
  B_ALGORITHM_OBJ     algorithmObject,                /* algorithm object */
  B_KEY_OBJ           keyObject,                             /* key object */
  B_ALGORITHM_CHOOSER algorithmChooser,              /* algorithm chooser */
  A_SURRENDER_CTX    *surrenderContext               /* surrender context */
);
```

Description

`B_KeyAgreeInit` initializes *`algorithmObject`* for performing key agreement using the algorithm specified by a previous call to `B_SetAlgorithmInfo`. The chooser for selecting the algorithm method is *`algorithmChooser`*. The key object for supplying the key information is *`keyObject`*; it should be `(B_KEY_OBJ)NULL_PTR` for key agreement algorithms that do not need an input key. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it.

This routine can be used for Diffie-Hellman key agreement. First `B_KeyAgreeInit` is called to setup the algorithm, then each party calls `B_KeyAgreePhase1` to generate the value that is then sent to the other party. Each party then passes the received value to `B_KeyAgreePhase2`, which computes the agreed-upon key. If several key agreements are done using the same algorithm, there is no need to call `B_KeyAgreeInit` again.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, ”Crypto-C Error Types”

B_KeyAgreePhase1

```
int B_KeyAgreePhase1 (
  B_ALGORITHM_OBJ  algorithmObject,                      /* algorithm object */
  unsigned char   *output,                             /* output data buffer */
  unsigned int    *outputLen,                       /* length of output data */
  unsigned int     maxOutputLen,               /* size of output data buffer */
  B_ALGORITHM_OBJ  randomAlgorithm,                    /* random byte source */
  A_SURRENDER_CTX *surrenderContext                     /* surrender context */
);
```

Description

B_KeyAgreePhase1 uses *algorithmObject* to generate the initial value for the other party, writing it to *output*, which is a buffer supplied by the caller of at least *maxOutputLen* bytes, and setting *outputLen* to the number of bytes written to *output*. The algorithm object for supplying random numbers is *randomAlgorithm*. The surrender context for processing and canceling during lengthy operations is *surrenderContext*; if its value is (A_SURRENDER_CTX *)NULL_PTR, Crypto-C does not use it. See B_KeyAgreeInit.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_KeyAgreePhase2

```
int B_KeyAgreePhase2 (
  B_ALGORITHM_OBJ  algorithmObject,                      /* algorithm object */
  unsigned char   *output,                             /* output data buffer */
  unsigned int    *outputLen,                        /* length of output data */
  unsigned int     maxOutputLen,               /* size of output data buffer */
  unsigned char   *input,                                      /* input data */
  unsigned int     inputLen,                          /* length of input data */
  A_SURRENDER_CTX *surrenderContext                       /* surrender context */
);
```

Description

B_KeyAgreePhase2 performs a round of key agreement as specified by *algorithmObject*, receiving *inputLen* bytes from *input*, which is the other party's intermediate value. B_KeyAgreePhase2 writes the output to *output*, which is a buffer supplied by the caller of at least *maxOutputLen* bytes, and sets *outputLen* to the number of bytes written to *output*. If *input* is the other party's final intermediate value, *output* is the agreed-upon key; otherwise, *output* is a new intermediate value to send to the other party. The surrender context for processing and canceling during lengthy operations is *surrenderContext*; if its value is (A_SURRENDER_CTX *)NULL_PTR, Crypto-C does not use it.

B_KeyAgreePhase2 may be called one or more times to process intermediate values, depending on how many other parties are involved in the key agreement. For an algorithm like Diffie-Hellman, B_KeyAgreePhase2 is called once.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_RandomInit

```
int B_RandomInit (
  B_ALGORITHM_OBJ     randomAlgorithm,          /* random algorithm object */
  B_ALGORITHM_CHOOSER algorithmChooser,               * algorithm chooser */
  A_SURRENDER_CTX    *surrenderContext                /* surrender context */
);
```

Description

B_RandomInit initializes *randomAlgorithm* for generating random bytes using the algorithm specified by a previous call to B_SetAlgorithmInfo. *randomAlgorithm* is ready to generate bytes after the call to B_RandomInit. However, it is necessary to mix in random seed values with B_RandomUpdate. Otherwise, without seed values, the bytes generated by the algorithm follow a default unseeded byte sequence. The chooser for selecting the algorithm method is *algorithmChooser*. The surrender context for processing and canceling during lengthy operations is *surrenderContext*; if its value is (A_SURRENDER_CTX *)NULL_PTR, Crypto-C does not use it.

B_RandomInit is called once to create the generator, then B_RandomUpdate is called one or more times to add “seed” bytes (values that are hard for an attacker to predict) to the generator. After enough seed is added, for example, at least 128 bytes, then B_GenerateRandomBytes can be called one or more times to generate blocks of pseudo-random data. If B_RandomUpdate is only called once before B_GenerateRandomBytes, then the BSAFE 2 algorithms will be used. Two or more calls to B_RandomUpdate will cause the improved BSAFE 3 algorithms to be used. It is also possible to call B_RandomUpdate after calling B_GenerateRandomBytes in order to add more hard-to-predict values to the generator. There is no need to call B_RandomInit again.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, ”Crypto-C Error Types”

B_RandomUpdate

```
int B_RandomUpdate (
  B_ALGORITHM_OBJ   randomAlgorithm,                  /* random algorithm */
  unsigned char    *input,                       /* block values to mix in */
  unsigned int     inputLen,                     /* length of input block */
  A_SURRENDER_CTX *surrenderContext                  /* surrender context */
);
```

Description

`B_RandomUpdate` mixes *`inputLen`* bytes from *`input`* into *`randomAlgorithm`*. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. See `B_RandomInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_SetAlgorithmInfo

```
int B_SetAlgorithmInfo (
  B_ALGORITHM_OBJ algorithmObject,                      /* algorithm object */
  B_INFO_TYPE     infoType,                /* type of algorithm information */
  POINTER         info                             /* algorithm information */
);
```

Description

B_SetAlgorithmInfo sets the parameters of *algorithmObject* to the information pointed to by *info*. The type of algorithm and the format of the parameters is specified by *infoType*, which is one of the algorithm info types with an AI_ prefix listed in Chapter 2. A separate copy of the information supplied by *info* is allocated inside the algorithm object so that *info* may be changed after the call to B_SetAlgorithmInfo. B_SetAlgorithmInfo returns BE_WRONG_ALGORITHM_INFO if the algorithm type encoded in *info* is not the type expected by *infoType.*

Once an algorithm object has been set, it should not be reset; that is, do not call B_SetAlgorithmInfo twice on a single created algorithm object. Either create a new algorithm object, or destroy an existing one and create it again.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_SetKeyInfo

```
int B_SetKeyInfo (
  B_KEY_OBJ   keyObject,                                    /* key object */
  B_INFO_TYPE infoType,                          /* type of key information */
  POINTER     info                                     /* key information */
);
```

Description

`B_SetKeyInfo` sets the value of *`keyObject`* to the information pointed to by *`info`*. The format of the information is specified by *`infoType`*, which is one of the key info types with a `KI_` prefix listed in Chapter 3. Also, some of the AI algorithm info types listed in Chapter 2 specify the key info type that should be used to set the key object needed by the algorithm. A separate copy of the information supplied by *`info`* is allocated inside the key object so that *`info`* may be changed after the call to `B_SetKeyInfo`. `B_SetKeyInfo` returns `BE_WRONG_KEY_INFO` if the key type encoded in *`info`* is not the type expected by *`infoType.`*

Once a key object has been set, it should not be reset; that is, do not call `B_SetKeyInfo` twice on a single created key object. Either create a new key object, or destroy an existing one and create it again.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_SignFinal

```
int B_SignFinal (
  B_ALGORITHM_OBJ  algorithmObject,                 /* algorithm object */
  unsigned char   *signature,                /* signature output buffer */
  unsigned int    *signatureLen,          /* length of signature output */
  unsigned int     maxSignatureLen,            /* size of output buffer */
  B_ALGORITHM_OBJ  randomAlgorithm,               /* random byte source */
  A_SURRENDER_CTX *surrenderContext                /* surrender context */
);
```

Description

`B_SignFinal` finalizes the digesting process for *`algorithmObject`* and computes the digital signature, writing the signature to *`signature`*, which is a buffer supplied by the caller of at least *`maxSignatureLen`* bytes, and sets *`signatureLen`* to the length of the signature. The algorithm object for supplying random numbers is *`randomAlgorithm`*; it may be `(B_ALGORITHM_OBJ)NULL_PTR` for signature algorithms that do not need random numbers. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. *`algorithmObject`* is reset to the state it was in after the call to `B_SignInit`, so that another signing process may be performed.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_SignInit

```
int B_SignInit (
  B_ALGORITHM_OBJ     algorithmObject,                /* algorithm object */
  B_KEY_OBJ           keyObject,                            /* key object */
  B_ALGORITHM_CHOOSER algorithmChooser,              /* algorithm chooser */
  A_SURRENDER_CTX    *surrenderContext               /* surrender context */
);
```

Description

`B_SignInit` initializes *`algorithmObject`* for computing a digital signature using the algorithm specified by a previous call to `B_SetAlgorithmInfo`. The chooser for selecting the algorithm method is *`algorithmChooser`*. The key object for supplying the key information is *`keyObject`*, which is typically a private key. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it.

`B_SignInit` only needs to be called once to set up a signature algorithm. The `B_SignUpdate` routine can be called multiple times to process blocks of data, and `B_SignFinal` is called once to process the last block, which includes producing the result. After `B_SignFinal` is called, `B_SignUpdate` can be called to start signing another sequence of blocks. There is no need to call `B_SignInit` again.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, ”Crypto-C Error Types”

B_SignUpdate

```
int B_SignUpdate (
  B_ALGORITHM_OBJ  algorithmObject,                    /* algorithm object */
  unsigned char    *partIn,                                   /* input data */
  unsigned int     partInLen,                       /* length of input data */
  A_SURRENDER_CTX *surrenderContext                     /* surrender context */
);
```

Description

`B_SignUpdate` updates the digesting process for *`algorithmObject`* with *`partInLen`* bytes from *`partIn`*. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. `B_SignUpdate` may be called zero or more times to supply the data by parts. See `B_SignInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	Error. See Appendix A, "Crypto-C Error Types."

B_SymmetricKeyGenerate

```
int B_SymmetricKeyGenerate (
 B_ALGORITHM_OBJ  algorithmObject,
 B_KEY_OBJ        symmetricKey,
 B_ALGORITHM_OBJ  randomObject,
 A_SURRENDER_CTX *surrenderContext;
);
```

Description

Creates a symmetric key in accordance with data specified during the `B_SetAlgorithmInfo` step. If hardware is present, the key information is stored in a `KI_TOKEN_INFO` structure. If no hardware is present, the key information is stored in `KI_EXTENDED_TOKEN_INFO` format, which extends the `KI_Token` base type.

Return value

Value	Description
0	Operation was successful.
nonzero	Error. See Appendix A, "Crypto-C Error Types."

B_SymmetricKeyGenerateInit

```
int B_SymmetricKeyGenerateInit (
 B_ALGORITHM_OBJ     algorithmObject,
 B_ALGORITHM_CHOOSER algorithmChooser,
 A_SURRENDER_CTX    *surrenderContext
);
```

Description

Initializes key generation object.

Return value

Value	Description
0	Operation was successful.
nonzero	error

B_VerifyFinal

```
int B_VerifyFinal (
  B_ALGORITHM_OBJ  algorithmObject,                    /* algorithm object */
  unsigned char   *signature,                       /* signature to verify */
  unsigned int     signatureLen,                    /* length of signature */
  B_ALGORITHM_OBJ  randomAlgorithm,                   /* random byte source */
  A_SURRENDER_CTX *surrenderContext                    /* surrender context */
);
```

Description

`B_VerifyFinal` finalizes the digesting process for *`algorithmObject`* and verifies the digital signature supplied by *`signature`* of *`signatureLen`* bytes. The algorithm object for supplying random numbers is *`randomAlgorithm`*; it may be `(B_ALGORITHM_OBJ)NULL_PTR` for signature algorithms that do not need random numbers. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. *`algorithmObject`* is reset to the state it was in after the call to `B_VerifyInit`, so that another verifying process may be performed. See `B_VerifyInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, ”Crypto-C Error Types”

B_VerifyInit

```
int B_VerifyInit (
  B_ALGORITHM_OBJ     algorithmObject,              /* algorithm object */
  B_KEY_OBJ           keyObject,                          /* key object */
  B_ALGORITHM_CHOOSER algorithmChooser,            /* algorithm chooser */
  A_SURRENDER_CTX    *surrenderContext             /* surrender context */
);
```

Description

`B_VerifyInit` initializes *`algorithmObject`* for verifying a digital signature using the algorithm specified by a previous call to `B_SetAlgorithmInfo`. The chooser for selecting the algorithm method is *`algorithmChooser`*. The key object for supplying the key information is *`keyObject`*, which is typically a public key. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it.

`B_VerifyInit` only needs to be called once to set up a verification algorithm. The `B_VerifyUpdate` routine can be called multiple times to process blocks of data, and `B_VerifyFinal` is called once to process the last block which includes checking the computed signature against the expected signature that is passed to `B_VerifyFinal`. After `B_VerifyFinal` is called, `B_VerifyUpdate` can be called to start verifying another sequence of blocks. There is no need to call `B_VerifyInit` again.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

B_VerifyUpdate

```
int B_VerifyUpdate (
  B_ALGORITHM_OBJ  algorithmObject,                    /* algorithm object */
  unsigned char   *partIn,                                   /* input data */
  unsigned int     partInLen,                      /* length of input data */
  A_SURRENDER_CTX *surrenderContext                    /* surrender context */
);
```

Description

`B_VerifyUpdate` updates the digesting process for *`algorithmObject`* with *`partInLen`* bytes from *`partIn`*. The surrender context for processing and canceling during lengthy operations is *`surrenderContext`*; if its value is `(A_SURRENDER_CTX *)NULL_PTR`, Crypto-C does not use it. `B_VerifyUpdate` may be called zero or more times to supply the data by parts. See `B_VerifyInit`.

Return value

Value	Description
0	Operation was successful.
non-zero	see Appendix A, "Crypto-C Error Types"

T_free

```
void T_free (
  POINTER block                                        /* block address */
);
```

Description

`T_free` deallocates *`block`*. The value of *`block`* is allocated with `T_malloc` or reallocated with `T_realloc`, or it is `NULL_PTR`. If *`block`* is `NULL_PTR`, `T_free` performs no operation.

Return value

There is no return value.

T_malloc

```
POINTER T_malloc (
  unsigned int len                                /* length of block */
);
```

Description

`T_malloc` allocates a memory block of at least *`len`* bytes. The value of *`len`* can be zero, in which case `T_malloc` returns a valid non-`NULL_PTR` value.

Return value

Value	Description
address of block	Operation was successful.
NULL_PTR	error

T_memcmp

```
int T_memcmp (
  POINTER      firstBlock,                             /* first block */
  POINTER      secondBlock,                           /* second block */
  unsigned int len                                  /* length of blocks */
);
```

Description

`T_memcmp` compares the first *`len`* bytes of *`firstBlock`* and *`secondBlock`*. The value of *`len`* can be zero, in which case *`firstBlock`* and *`secondBlock`* are undefined and `T_memcmp` returns zero. `T_memcmp` compares the blocks by scanning the blocks from lowest address to highest until a difference is found. The smaller-valued block is the one with the smaller-valued byte at the point of difference. If no difference is found, then the blocks are equal.

Return value

Value	Description
< 0	*`firstBlock`* is smaller.
0	The blocks are equal.
> 0	*`firstBlock`* is larger.

T_memcpy

```
void T_memcpy(
  POINTER      output,                                    /* output block */
  POINTER      input,                                      /* input block */
  unsigned int len                                   /* length of blocks */
);
```

Description

`T_memcpy` copies the first *`len`* bytes of *`input`* to *`output`*. The value of *`len`* can be zero, in which case *`output`* and *`input`* are undefined. The blocks do not overlap.

Return value

There is no return value.

T_memmove

```
void T_memmove (
  POINTER      output,                                     /* output block */
  POINTER      input,                                       /* input block */
  unsigned int len                                      /* length of blocks */
);
```

Description

`T_memmove` copies the first *`len`* bytes of *`input`* to *`output`*. The blocks can overlap. The value of *`len`* can be zero, in which case *`output`* and *`input`* are undefined.

Return value

There is no return value.

T_memset

```
void T_memset (
  POINTER      output,                               /* output block */
  int          value,                                       /* value */
  unsigned int len                                /* length of block */
);
```

Description

`T_memset` sets the first *`len`* bytes of *`output`* to *`value`*. If the value of *`len`* is zero, *`output`* is undefined.

Return value

There is no return value.

T_realloc

```
POINTER T_realloc (
  POINTER      block,                                         /* block address */
  unsigned int len                                               /* new length */
);
```

Description

`T_realloc` changes the size of *`block`* to *`len`*. It allocates a memory block of length *`len`* bytes, copies as many bytes as possible from the old memory block to the new one, and frees the old block. The address of the new block can be different from the address of the old block. The value of *`len`* can be zero, in which case `T_realloc` returns a valid non-`NULL_PTR` value. On error, *`block`* is freed. Note that many implementations of `realloc` do not free the block on error, so `T_realloc` must to do this. The value of *`block`* is allocated with `T_malloc` or reallocated with `T_realloc`, or it is `NULL_PTR`. If *`block`* is `NULL_PTR`, `T_realloc` performs as `T_malloc`.

Return value

Value	Description
address of new block	Operation was successful.
NULL_PTR	error

T_strcmp

```
int T_strcmp (
  const char *string1,                                  /* first string */
  const char *string2                                  /* second string */
);
```

Description

`T_strlen` compares two strings. The return value indicates the lexicographic relation of *string1* to *string2*.

Return Value

Value	Relationship of ***string1*** to ***string2***
< 0	*string1* is less than *string2*
0	*string1* is identical to *string2*
> 0	*string1* is greater than *string2*

T_strcpy

```
void T_strcpy (
     char *strDest,                                      /* destination string */
     char *strSource                                          /* source string */
);
```

The `T_strcpy` function copies *`strSource`*, including the terminating `NULL` character, to the location specified by *`strDest`*.

Return Value

The destination string.

T_strlen

```
unsigned int T_strlen (
     char *pStr                                  /* null-terminated string */
);
```

Description

T_strlen returns the number of characters in *pStr*, excluding the terminal NULL.

Return Value

The number of characters in the string.

Appendix A

Crypto-C Error Types

This appendix lists the RSA BSAFE Crypto-C (Crypto-C) error types.

Table A-1 **Crypto-C Error Types**

Hex	Decimal	Error Code	Description
0x0200	512	BE_ALGORITHM_ALREADY_SET	the value of the algorithm object has already been set by a call to `B_SetAlgorithmInfo` or by an algorithm parameter generation
0x0201	513	BE_ALGORITHM_INFO	invalid format for the algorithm information in the algorithm object
0x0202	514	BE_ALGORITHM_NOT_INITIALIZED	algorithm object has not been initialized by a call to the Init procedure
0x0203	515	BE_ALGORITHM_NOT_SET	the algorithm object has not been set by a call to `B_SetAlgorithmInfo`
0x0204	516	BE_ALGORITHM_OBJ	invalid algorithm object
0x0205	517	BE_ALG_OPERATION_UNKNOWN	unknown operation for an algorithm or algorithm info type
0x0206	518	BE_ALLOC	insufficient memory
0x0207	519	BE_CANCEL	operation was cancelled by the surrender function
0x0208	520	BE_DATA	generic data error

Table A-1 **Crypto-C Error Types**

Hex	Decimal	Error Code	Description
0x0209	521	BE_EXPONENT_EVEN	invalid even value for public exponent in keypair generation
0x020a	522	BE_EXPONENT_LEN	invalid exponent length for public exponent in keypair generation
0x020b	523	BE_HARDWARE	cryptographic hardware error
0x020c	524	BE_INPUT_DATA	invalid encoding format for input data
0x020d	525	BE_INPUT_LEN	invalid total length for input data
0x020e	526	BE_KEY_ALREADY_SET	the value of the key object has already been set by a call to `B_SetKeyInfo` or by a key generation
0x020f	527	BE_KEY_INFO	invalid format for the key information in the key object
0x0210	528	BE_KEY_LEN	invalid key length
0x0211	529	BE_KEY_NOT_SET	the key object has not been set by a call to `B_SetKeyInfo` or by a key generation
0x0212	530	BE_KEY_OBJ	invalid key object
0x0213	531	BE_KEY_OPERATION_UNKNOWN	unknown operation for a key info type
0x0214	532	BE_MEMORY_OBJ	invalid internal memory object
0x0215	533	BE_MODULUS_LEN	unsupported modulus length for a key or for algorithm parameters
0x0216	534	BE_NOT_INITIALIZED	algorithm is improperly initialized
0x0217	535	BE_NOT_SUPPORTED	the algorithm chooser does not support the type of key information in the key object for the specified algorithm
0x0218	536	BE_OUTPUT_LEN	the maximum size or the output buffer is too small to receive the output
0x0219	537	BE_OVER_32K	data block exceeds 32,767 bytes
0x021a	538	BE_RANDOM_NOT_INITIALIZED	the random algorithm has not been initialized by a call to `B_RandomInit`
0x021b	539	BE_RANDOM_OBJ	invalid algorithm object for the random algorithm
0x021c	540	BE_SIGNATURE	signature does not verify

Table A-1 **Crypto-C Error Types**

Hex	Decimal	Error Code	Description
0x021d	541	BE_WRONG_ALGORITHM_INFO	the required algorithm information is not in the algorithm object
0x021e	542	BE_WRONG_KEY_INFO	the required key information is not in the key object
0x021f	543	BE_INPUT_COUNT	Update called an invalid number of times for inputting data
0x0220	544	BE_OUTPUT_COUNT	Update called an invalid number of times for outputting data
0x0221	545	BE_METHOD_NOT_IN_CHOOSER	algorithm chooser doesn't contain the algorithm method for the algorithm specified by the previous call to `B_SetAlgorithmInfo`
0x0222	546	BE_KEY_WEAK	the key data supplied would generate a known weak key
0x0223	547	BE_EXPONENT_ONE	the value of the public exponent cannot be 1
0x0224	548	BE_BAD_POINTER	invalid pointer
0x0225	549	BE_BAD_PASSPHRASE	invalid password
0x0226	550	BE_AM_DOMESTIC_ONLY	an attempt was made to call a function that is not available in the export version of Crypto-C
0x0227	551	BE_BAD_SEEDING	bad seeding was passed to an AI_X931Random object

Appendix B

Platform-Specific Types and Constants

This appendix lists the platform-specific types and constants.

Types

Crypto-C requires these platform-specific types: `POINTER`, `UINT2`, and `UINT4`. These can be found in the file `aglobal.h`.

POINTER

A `POINTER` value is a generic pointer to memory to which any other pointer can be cast.

Example:

```
typedef unsigned char *POINTER;
```

UINT2

A `UINT2` value is a 16-bit unsigned integer.

Example:

```
typedef unsigned short int UINT2;
```

UINT4

A `UINT4` value is a 32-bit unsigned integer.

Example:

```
 typedef unsigned long int UINT4;
```

Constants

Crypto-C requires one macro: PROTO_LIST.

PROTO_LIST indicates the form that C function prototypes are to take. If function prototypes specify the types of the arguments, then PROTO_LIST should be defined as:

```
#define PROTO_LIST(list) list
```

Otherwise PROTO_LIST should be defined as:

```
#define PROTO_LIST(list) ()
```

Crypto-C defines one string constant: BSAFE_VERSION:

```
extern char *BSAFE_VERSION;
```

BSAFE_VERSION is a NULL-terminated string constant that specifies the release of the Crypto-C library.

Appendix C

References

FIPS PUB 46-1	National Bureau of Standards. *FIPS Publication 46-1: Data Encryption Standard.* January 1988.
FIPS PUB 81	National Bureau of Standards. *FIPS Publication 81: DES Modes of Operation.* December 1980.
FIPS PUB 180-1	National Institute of Standards and Technology. *FIPS Publication 180-1: Secure Hash Standard.* May 1993.
FIPS PUB 186	National Institute of Standards and Technology. *FIPS Publication 186: Digital Signature Standard.* May 1994.
P1363 Draft D1	IEEE. *Standard Specifications for Public Key Cryptography.* December 1997.
RFC 1113	J. Linn. *RFC 1113: Privacy Enhancement for Internet Electronic Mail: Part I Message Encipherment and Authentication Procedures.* August 1989.
RFC 1319	B. Kaliski. *The MD2 Message-Digest Algorithm.* April 1992.
RFC 1321	R. Rivest. *The MD5 Message-Digest Algorithm.* April 1992.
RFC 1423	D. Balenson. *RFC 1423: Privacy Enhancement for Internet Electronic Mail: Part III Algorithms, Modes, and Identifiers*. February 1993.
X.208	CCITT. *Recommendation X.208: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).* 1988.

X.209	CCITT. *Recommendation X.209: Specification of Abstract Syntax Notation One (ASN.1)*. 1988.
X.509	CCITT. *Recommendation X.509: The Directory Authentication Framework.* 1988.
X9.31	*Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA)*. December 1997.
X9.44 Draft	*Key management using reversible public key cryptography for the financial services industry.* January 1998.
X9.52 Draft	*Triple Data Encryption Algorithm Modes of Operation*. December 1997.
X9.57	ANSI. *Certificate Management,* N5-95, June 15, 1995.
X9.62	ANSI. *The Elliptic Curve Digital Signature Algorithm (ECDSA).* August 29, 1997.
X9.63	*Public Key Cryptography for the Financial Services Industry: Elliptic Curve Key Agreement and Transport Protocols.* December 1997.
[NIST91]	NIST. *Special Publication 500-202: Stable Implementation Agreements for Open Systems Interconnection Protocols.* Version 5, Edition 1, Part 12. December 1991.
	S.C. Kothari. *Proceedings of CRYPTO 84: Generalized Linear Threshold Scheme.* 1984.

The following references are from RSA Security Inc.'s Public-Key Cryptography Standards (PKCS) suite:

PKCS #1	RSA Security Inc. *PKCS #1: RSA Encryption Standard.* Version 1.5, November 1993.
PKCS #3	RSA Security Inc. *PKCS #3: Diffie-Hellman Key-Agreement Standard.* Version 1.4, November 1993.
PKCS #5	RSA Security Inc. *PKCS #5: Password-Based Encryption Standard.* Version 1.5, November 1993.
PKCS #7	RSA Security Inc. *PKCS #7: Cryptographic Message Syntax Standard.* Version 1.5, November 1993.
PKCS #8	RSA Security Inc. *PKCS #8: Private-Key Information Syntax Standard.* Version 1.2, November 1993.

PKCS documents are available at `http://www.rsasecurity.com/rsalabs/pkcs/`.

Index